

152

福建省高校计算机系列教材

TP 312C-93
Y21

C 语言程序设计 与应用教程

主 编 严桂兰

编写者 (以姓氏笔画为序)

黄思先

彭 洪



A0964753

厦门大学出版社

图书在版编目(CIP)数据

C 语言程序设计与应用教程/严桂兰主编. —厦门:厦门大学出版社, 2001. 8
ISBN 7-5615-1779-3

I. C… I. 严… III. C 语言-程序设计-水平考试-教材 IV. TP312

中国版本图书馆 CIP 数据核字(2001)第 034265 号

厦门大学出版社出版发行

(地址:厦门大学 邮编:361005)

<http://www.xmupress.com>

xmup@public.xm.fj.cn

福建沙县方圆印刷有限公司印刷

2001 年 8 月第 1 版 2002 年 1 月第 2 次印刷

开本:787×1092 1/16 印张:27.5

字数:700 千字 印数:2 501—6 000 册

定价:38.00 元

本书如有印装质量问题请直接寄承印厂调换

内容简介

福建省一批资深的教师,他们长期从事着 C 语言教学与科研,又多年参加福建省 C 语言二级等级考试命题,对 C 语言的内涵、规律有着独到的见解,他们根据自己的经验,以研讨学术的态度编写了本书。

在书的内容组织上,除按常规讲授 C 语言的基本、一般的内容外,还依逻辑思维方式将内容归类,如在数组、指针、函数的基本内容之后,开设一章来描述三者的简单应用;为了拓宽知识面,本书讲授了图形、调用中断方面的基本应用;为了上机需要,本书在有关章节安排了上机步骤、程序调试与出错信息;最后,还对 VC++ 作了简单介绍,它将 C 与 VC++ 连贯一气,顺理成章。本书在同类书中具有全面、应用性强、概念清晰等诸多特点。

本书可作为大专院校计算机与其他各类专业的教材,同时,也可供各行各业从事计算机工作的人员使用。

前 言

我们受福建省计算机等级考试指导委员会的委托,编写一本结合我省具体情况的C语言教材。虽然我们长期从事着C语言的教学与科研工作,又多年参加福建省C语言二级等级考试的命题,对C语言的内涵、规律的掌握有着一定优势,但要做到“统观全局,展望未来”、“深入浅出,突出实用”却不是那么轻松的。我们根据自己的经验,以研讨学术的态度,认真吸取各家的长处,利用集体的智慧,尽量以最好的质量来回报福建省各高校师生及广大读者。

首先,我们全面展开C语言的各知识点,但侧重不一。核心部分仍为“数组、函数、指针”,它们既是本书的重点,也是本书的难点。我们本着复杂问题简单化、简单问题实用化的思路,在除讲授数组、指针、函数的基本内容之外,还专门开设第七章来描述三者的简单应用,以让读者进一步掌握。

其次,我们以层层深化的写法,将各章节的本质与内涵呈现在读者的面前。因此,本书的另一特点就是层次分明,描述细化。

C语言是一门实践性很强的、且十分灵活的高级程序设计语言,其上机环节十分重要,为此,我们对上机步骤、程序调试等方面的内容作了具体章节安排,以突出其作用。

为了拓宽知识面,本书讲授了图形、调用中断并对C++进行了简单的介绍,这些内容并不是教纲所要求的,只作为课外参考内容(以*标示)。在时间安排上,我们建议课堂讲授、上机与自学并进的方式。虽然本书篇幅稍多,但很多部分是留给读者自学的,如附录、部分例子、以*标示的章节等。建议课堂讲授时间至少为54学时,上机为0.8~1倍的课堂讲授时间,自学为1~1.5倍的课堂讲授时间。

本书由华侨大学严桂兰教授主编,其中第1、2、4、5、11章由福建农林大学黄思先老师编写,第3、6、8、9章由厦门大学彭洪老师编写,第7、10章由华侨大学严桂兰教授编写,最后,由福州大学刘传才博士审阅。编写期间,多次得到福建省教育厅高教处、福建省计算机等级考试指导委员会、厦门大学出版社的大力支持与帮助;福建农林大学吴锤红教授、华侨大学陈启泉教授等提出了中肯而宝贵的意见。在此,一一致以衷心的感谢。

由于时间仓促,书中不妥之处在所难免,敬请多多批评指正。

编 者

2001年5月

目 录

前言

第一章 C 语言特点与上机操作	(1)
1.1 C 语言特点	(1)
1.1.1 C 语言的产生与发展	(1)
1.1.2 计算机语言与程序设计	(1)
1.1.3 C 语言的特点	(2)
1.2 C 语言程序基本组成	(2)
1.3 Turbo C 2.0 上机步骤	(5)
1.3.1 编辑、编译、连接、执行及调试程序的概念	(5)
1.3.2 Turbo C 的上机步骤	(6)
习题	(7)
第二章 C 语言的语法基础	(8)
2.1 基本数据类型	(8)
2.1.1 标识符与基本数据类型	(8)
2.1.2 常量与变量	(10)
2.1.3 内存的概念	(15)
2.2 基本输入、输出函数	(16)
2.2.1 格式输入函数和格式输出函数	(17)
2.2.2 非格式化输入、输出函数	(26)
2.3 运算符与表达式	(28)
2.3.1 算术运算	(29)
2.3.2 增 1 与减 1 运算	(30)
2.3.3 关系、逻辑及条件运算	(31)
2.3.4 位运算	(33)
2.3.5 赋值运算	(35)
2.3.6 类型转换	(39)
2.3.7 逗号运算	(39)
2.3.8 长度运算符	(39)
2.4 小结	(40)
习题	(40)

第三章 程序控制结构	(45)
3.1 C 语言的语句	(45)
3.2 顺序结构	(46)
3.3 分支结构	(47)
3.3.1 if 结构	(47)
3.3.2 switch 结构	(51)
3.4 循环结构	(54)
3.4.1 当型循环(前判定循环)	(54)
3.4.2 直到型循环(后判定循环)	(58)
3.4.3 break 语句与 continue 语句	(61)
3.5 goto 语句与标号	(65)
习题	(67)
第四章 构造型数据类型	(71)
4.1 数组	(71)
4.1.1 一维数组	(71)
4.1.2 字符数组	(78)
4.1.3 二维数组	(81)
4.2 结构体	(86)
4.2.1 结构体的概念	(86)
4.2.2 结构体类型及结构体变量	(87)
4.2.3 结构体变量的使用	(89)
4.2.4 结构体变量、结构体数组的初始化	(93)
4.2.5 位段	(95)
4.3 共用体	(96)
4.3.1 共用体的概念、类型说明和变量定义	(96)
4.3.2 共用体变量的使用	(99)
4.4 枚举型	(101)
4.5 typedef 的用途	(102)
4.6 小结	(103)
习题	(104)
第五章 指针	(112)
5.1 指针与指针变量	(112)
5.1.1 指针的基本概念	(112)
5.1.2 指针变量的定义	(113)
5.1.3 指针变量的赋值	(114)
5.2 指针运算符	(116)
5.2.1 指针运算符	(116)

5.2.2	无类型指针	(117)
5.3	指针与一维数组	(118)
5.3.1	指针与一维数组	(118)
5.3.2	移动指针及两指针相减运算	(120)
5.3.3	指针比较	(121)
5.3.4	字符串	(123)
5.3.5	指针与二维数组	(127)
5.4	指向指针的指针	(135)
5.4.1	指向指针的指针	(135)
5.4.2	定义指向指针的指针变量	(136)
5.4.3	指向指针的指针变量的应用	(137)
5.5	指针与结构	(138)
5.5.1	指向结构体变量的指针变量	(138)
5.5.2	指向结构体数组的指针变量	(139)
5.5.3	通过指针变量存取位段数据	(141)
5.6	指向共用体和枚举型的指针	(141)
5.6.1	指向共用体变量的指针变量	(141)
5.6.2	指向枚举型的指针变量	(143)
5.7	指针小结	(143)
5.7.1	指针概念综述	(143)
5.7.2	指针运算小结	(144)
5.7.3	等价表达式	(145)
习题		(146)
第六章	函数	(154)
6.1	常见的系统库函数	(154)
6.1.1	字符与字符串函数	(155)
6.1.2	简单数学函数	(158)
6.1.3	类型转换函数	(160)
6.2	用户自定义函数	(161)
6.2.1	函数定义、调用和说明	(161)
6.2.2	函数返回值	(164)
6.2.3	函数参数	(165)
6.3	函数的嵌套调用及递归调用	(166)
6.3.1	函数的嵌套调用	(166)
6.3.2	函数的递归调用	(167)
6.4	局部变量与全局变量	(174)
6.5	变量的存储类型与变量的初始化	(176)
6.6	外部函数与内部函数	(180)
6.7	编译预处理	(185)

6.7.1 文件包含	(185)
6.7.2 宏定义	(185)
6.7.3 条件编译	(187)
习题	(188)
第七章 数组、指针、函数的应用	(191)
7.1 概述	(191)
7.2 函数之间的数据传递	(193)
7.2.1 函数数据按数值传递	(193)
7.2.2 函数数据按地址传递	(194)
7.2.3 利用函数返回值和外部变量进行函数数据传递	(203)
7.2.4 结构作为函数参数传递	(204)
7.3 函数指针与指针函数	(207)
7.3.1 函数指针	(207)
7.3.2 指针函数	(209)
7.4 数组指针、指针数组与带参的 main 函数	(211)
7.4.1 数组指针	(211)
7.4.2 指针数组	(211)
7.4.3 带参的 main 函数	(212)
7.5 单向链表	(214)
7.5.1 单向链表的概念	(214)
7.5.2 链表的建立	(214)
7.5.3 链表结点的删除	(217)
7.5.4 链表结点的插入	(218)
7.6 小结	(220)
习题	(221)
第八章 文件	(229)
8.1 文件、流和文件系统	(229)
8.2 缓冲文件系统	(230)
8.2.1 文件的打开、关闭和文件结束测试	(231)
8.2.2 文件的读写	(232)
8.2.3 文件的定位	(237)
8.2.4 出错的处理	(240)
8.3 非缓冲文件系统	(240)
习题	(243)
* 第九章 实用程序设计初步	(246)
9.1 图形处理	(246)
9.1.1 图形模式的初始化	(246)

9.1.2	独立图形运行程序的建立	(249)
9.1.3	屏幕颜色的设置和清屏函数	(249)
9.1.4	基本图形函数	(251)
9.1.5	填充	(254)
9.1.6	图形窗口和图形屏幕操作函数	(257)
9.1.7	图形模式下的文本输出	(259)
9.2	中断处理	(263)
9.2.1	中断的允许和禁止	(264)
9.2.2	DOS 与 BIOS 功能调用	(264)
9.2.3	中断服务程序	(269)
* 第 10 章	C++ 简介	(271)
10.1	C++ 的新特征	(271)
10.1.1	C++ 的输入/输出	(271)
10.1.2	C++ 的函数原型	(273)
10.1.3	C++ 函数的缺省参数	(273)
10.1.4	C++ 的 new 与 delete	(274)
10.1.5	C++ 的内联函数	(275)
10.1.6	C++ 的引用	(276)
10.1.7	C++ 面向对象编程基础	(277)
10.2	C++ 编程的核心技术	(279)
10.2.1	类的定义与使用	(279)
10.2.2	数据的封装	(281)
10.2.3	函数的重载	(283)
10.2.4	对象的初始化	(283)
10.2.5	缺省构造函数、拷贝构造函数与析构函数	(285)
10.3	类成员与对象的构造	(287)
10.3.1	使用 this 指针	(287)
10.3.2	使用静态成员	(289)
10.3.3	使用友员	(293)
10.3.4	使用对象成员	(294)
10.3.5	使用对象数组	(296)
10.3.6	使用指向对象的指针	(298)
10.3.7	类型的转换	(299)
10.4	派生类的构造	(300)
10.4.1	派生类的定义	(300)
10.4.2	类的保护成员	(302)
10.4.3	访问权限的设置	(303)
10.4.4	派生类的构造函数与析构函数	(305)
10.4.5	多重继承	(307)

10.4.6	在派生类中改写基类的成员函数	(308)
10.4.7	虚拟函数	(310)
10.4.8	纯虚拟函数与抽象类	(311)
10.5	运算符重载	(314)
10.5.1	运算符重载的作用与形式	(314)
10.5.2	类运算符与友元运算符	(315)
10.5.3	++与--运算符的重载	(317)
10.5.4	重载 new 和 delete	(320)
10.6	输入/输出流的使用	(320)
10.6.1	标准的屏幕输出	(321)
10.6.2	标准的键盘输入	(327)
10.6.3	用户自定义类的输入/输出	(329)
10.6.4	格式化字符串流类的使用	(331)
10.6.5	磁盘文件的输入/输出	(333)
10.6.6	打印机的使用	(342)
10.7	模板	(343)
10.7.1	模板的基本概念	(343)
10.7.2	函数模板的定义与使用	(344)
10.7.3	类模板的定义与使用	(346)
习题		(348)
第十一章	Turbo C 集成开发环境中调试程序	(353)
11.1	Turbo C 集成开发环境调试程序基本要领	(353)
11.1.1	纠正编译错误	(353)
11.1.2	纠正连接错误	(357)
11.1.3	纠正逻辑错误	(358)
11.2	调试程序实例	(362)
11.3	调试程序命令和热键小结	(367)
11.4	Turbo C 程序的常见错误	(368)
11.4.1	使用变量容易出现的错误	(368)
11.4.2	编写表达式容易出现的错误	(368)
11.4.3	使用语句容易出现的错误	(370)
11.4.4	使用数组容易出现的错误	(372)
11.4.5	使用库函数容易出现的错误	(373)
11.4.6	使用自定义函数容易出现的错误	(374)
11.4.7	使用指针变量容易出现的错误	(377)
11.4.8	其他常见错误	(378)
11.5	小结	(379)
习题		(379)

附录 A	C 语法摘要	(380)
附录 B	数值系统	(385)
附录 C	Turbo C 2.0 集成开发环境的使用	(390)
附录 D	ASCII 字符集	(402)
附录 E	运算符的优先级与结合性	(404)
附录 F	Turbo C 的部分标准函数	(405)
附录 G	编译错误信息	(409)
附录 H	习题参考答案	(417)

第 1 章

C 语言特点与上机操作

学习计算机程序设计语言是提高人们计算机知识水平的重要步骤。C 语言作为当今最为流行的程序设计语言之一,不但成为计算机专业的必修课程,而且也越来越多地成为非计算机专业的学习课程。本章介绍 C 语言的发展与特点,叙述 C 语言程序的组成与结构,阐明 C 语言的上机步骤和方法。

建议本章授课 2 学时,上机 2 学时,自学 3 学时。

1.1 C 语言特点

1.1.1 C 语言的产生与发展

C 语言是 1971 年由美国贝尔实验室的 D. M. Ritchie 用了一年的时间设计发明的,1972 年投入使用。1973 年 K. Thompson 和 D. M. Ritchie 用 C 语言重写 UNIX 操作系统获得了巨大成功。随着微型计算机的日益普及,出现了许多 C 语言版本。1983 年美国国家标准化协会 (ANSI) 为 C 语言制定了一套 ANSI 标准,1987 年 ANSI 公布的 87 ANSI 标准成为现行的 C 语言标准。90 年代至今,美国 Borland 公司陆续推出了 Turbo C、Turbo C++、Borland C++ 以及 C++ Builder 等系列产品,Microsoft 公司也推出了 Microsoft C、Visual C 等产品。目前这些产品均提供了面向对象的可视化开发环境,用户可以快速、方便地建立 DOS/Windows 应用程序。C 语言已成为程序员使用最多的编程语言之一。无论是面向硬件编程,还是面向大型数据库编程,无论是编写应用软件,还是编写操作系统,C 语言都是首选编程语言。

本书内容以 Turbo C 2.0 为标准。

1.1.2 计算机语言与程序设计

计算机要完成某一特定的任务,必须执行一系列计算机指令。程序就是由这样的一系列计算机指令组成的。程序设计就是针对某一要处理的问题,设计出解决该问题的计算机指令序列。因此程序设计是一项创造性的工作。进行程序设计必须借助语言来描述,这些用来描述的语言就是程序设计语言。只有严格按照程序设计语言的语法规则来书写程序,才能让计算机正确执行指令序列,完成指定的任务。程序设计语言分为低级语言和高级语言两大类。低级语言直接面向机器,如机器语言和汇编语言;高级语言独立于机器,用高级语言编写的程序在不同的机器上必须使用不同的翻译程序。C 语言程序是一种高级语言程序,它必须被翻译成计算机

能识别的语言,即机器语言,才能在计算机上运行。

1.1.3 C 语言的特点

C 语言之所以能迅速崛起,并成为最受欢迎的程序设计语言之一,是因为它有许多优于其他语言的特点。C 语言具有下列特点:

1. C 语言功能齐全

C 语言的数据类型有:整型、实型、字符型、无符号整型、数组类型、指针类型、结构体类型、共用体类型、枚举型等。C 语言运算符丰富,表达式类型有:赋值表达式、算术表达式、关系表达式、逻辑表达式、条件表达式、逗号表达式以及位运算等。

2. C 语言简洁、紧凑,使用方便、灵活

C 语言的一个语句可完成多项操作,一个表达式也可以完成多项操作。书写简练,源程序短,因而输入程序的工作量小。

3. C 是面向结构化程序设计的语言

结构化语言的显著特点是代码、数据的模块化,C 程序是以函数形式提供给用户的,这些函数调用方便。C 语言具有多种条件语句、循环控制程序流向语句(如 if /else 语句、switch 语句、while 语句、do/while 语句、for 语句、break 语句、continue 语句等),从而使程序完全结构化。

4. C 是中级语言

C 语言把高级语言的基本结构和语句与低级语言的实用性结合起来。C 语言可以像汇编语言一样对位、字节和地址进行操作,实现汇编语言的大部分功能,可直接对硬件进行编程。用 C 语言加上一些汇编语言子程序编程,更能显示 C 语言的优势。C 语言源程序编译后代码短,执行效率高。

5. C 语言适用范围大

C 语言还有一个突出的优点就是适合于多种操作系统,如 DOS、Windows、UNIX,也适用于多种机型。源程序代码可移植性好。

1.2 C 语言程序基本组成

下面是两个 C 语言程序的例子:

[例 1.1] 由键盘输入三角形的三个边长,计算出该三角形的面积。

```
#include "stdio.h"    /* 文件包含,输入、输出函数 */
#include "math.h"     /* 文件包含,数学函数 */
main()               /* 主函数 */
{
    float a, b, c, l, area; /* 定义局部变量 */
    printf("请输入三角形三条边的边长:");
    scanf("%f%f%f", &a, &b, &c); /* 由键盘输入三角形三边的边长 */
    l = (a+b+c)/2.0;
    area = sqrt(l*(l-a)*(l-b)*(l-c)); /* 函数 sqrt(x)是求 x 的平方根 */
}
```

```
printf("该三角形的面积是:%6.2f\n", area);
}
```

运行结果:

请输入三角形三条边的边长:3.4 5.6 7.4

该三角形的面积是: 9.05

[例 1.2] 与[例 1.1]一样,由键盘输入三角形的三个边长,计算出该三角形的面积。但使用函数来求面积。

```
#include "stdio.h"
#include "math.h"
float triangle _ area(float a, float b, float c) /* 定义用户函数 */
{
    float l;
    l=(a+b+c)/2.0;
    return sqrt(l*(l-a)*(l-b)*(l-c));
}
main() /* 主函数定义 */
{
    float a, b, c, area;
    printf("请输入三角形三条边的边长:");
    scanf("%f%f%f", &a, &b, &c);
    area=triangle _ area(a,b,c);
    printf("该三角形的面积是:%6.2f\n", area);
}
```

运行结果与[例 1.1]相同。

从上面例子可以看出,C 程序的组成主要有以下几个特点:

1. 一个 C 源程序由函数构成,其中必须有且只能有一个主函数(main 函数),还可以有 0 至多个其他函数。C 程序由 main 函数的首句开始执行,由 main 函数的最后一句结束,函数中可调用其他函数。在 C 语言中函数分为两种,用户可以自己定义函数(如[例 1.2]中的 triangle _ area 函数),也可以使用 C 语言系统提供的库函数(如 printf 函数和 scanf 函数)。Turbo C 提供了 300 多个库函数,要调用 C 的库函数,必须在源程序首部加上相应的库文件包含(如上述例子中的 #include "stdio.h")。

2. C 程序一般用小写字母书写,大、小写字母是有区别的,如 area 与 Area 代表不同的变量。C 程序书写格式自由,一行内可写多条语句,若一条语句较长,可分写在多行上。一般情况下语句中的空格和回车符可忽略不计。语句用分号“;”结尾,分号“;”是 C 语句的一部分。可以在{ }内写若干条语句,构成复合语句。用 C 语言编程时,建议一行写一条语句,遇到复合语句向右缩进,必要时对程序加上注释行。这样写出的源程序结构清楚,易于阅读、调试、维护和修改。

3. C 语言的变量在使用之前必须先定义其数据类型,未经定义的变量不能使用。一般应在可执行语句前面定义变量类型。

4. 函数由函数头与函数体两部分组成。第一部分为函数头(函数说明部分),包括函数返

返回值类型、函数名、函数参数及参数的数据类型。第二部分为函数体部分,它是函数功能的实现部分,包括变量定义与执行语句。

5. 一个较完整的程序通常包括:文件包含(一组 #include 语句)、用户函数说明部分、全局变量定义、主函数和若干用户函数。在主函数和用户函数中又包括局部变量定义、若干个 C 库函数调用语句、控制流程语句、用户函数的调用语句等。C 源程序的一般形式为:

```
文件包含
用户函数说明
全局变量定义
main()
{
    局部变量定义语句
    语句
}
fun1()
{
    局部变量定义语句
    语句
}
fun2()
{
    局部变量定义语句
    语句
}
    :
funN()
{
    局部变量定义语句
    语句
}
```

其中 fun1(), fun2(), ..., funN() 代表用户定义的函数,语句指赋值语句、控制流程语句、C 提供的任何库函数调用语句或用户函数调用语句等。C 语言函数内部不能定义函数,函数之间是平等的。主函数 main 可以放在某一用户函数之前,也可以放在某一用户函数之后,但被调用的函数应在主调函数之前定义或说明。在编写较大型的 C 程序时,常把源程序分成多个文件编写,采用文件包含或工程文件(即项目文件 *.PRJ)的方法连接成可执行程序(请参阅附录 C“Turbo C 2.0 集成开发环境的使用”)。

6. 用户为了提高源程序的可读性,可在 C 程序中加上注释部分,编译时注释部分被滤掉。C 程序的注释部分包含在“/*”和“*/”之间,/和*之间不允许有空格。注释部分允许出现在程序中的任何位置。

1.3 Turbo C 2.0 的上机步骤

1.3.1 编辑、编译、连接、执行及调试程序的概念

1. 编辑

程序员用 C 语言编写的程序称为 C 的源程序(一般为 *.C 文件)。编辑就是编写源程序的过程,它包括新建一个源程序文件或修改已有的源程序文件,它的操作有插入、删除、修改源程序。除了 Turbo C 2.0 集成开发环境能够编辑源程序外,还可使用 DOS 环境中的 EDIT、CCED、WPS 或 Windows 环境中的 WORD、记事本、写字板等常用的编辑软件来编辑 C 的源程序,存盘时应采用纯文本方式保存文件。

2. 编译

源程序是以纯文本形式存储的,必须翻译成机器语言才能被计算机识别。完成这一翻译工作的就是所谓的编译程序。源程序经过编译程序翻译成等价的机器语言程序——目标程序(一般为 *.OBJ 文件),这一翻译过程称为编译。Turbo C 2.0 集成开发环境带有编译程序。

3. 连接

如果编译成功,还应将目标程序和 C 的库函数连接成可执行程序(一般为 *.EXE 文件),并存储在计算机的存储设备(外存)中,以便执行。负责目标程序和库函数连接工作的程序称为连接程序。Turbo C 2.0 集成开发环境带有连接程序。

4. 执行

源程序经过编译、连接成为可执行文件(扩展名为 .exe 或 .com)后,一般存于计算机的外存中。所谓执行程序就是把可执行文件从外存调入计算机内存,并由计算机完成该程序预定的功能,如完成输入数据,处理数据及输出结果等任务。执行程序又称为运行程序。

5. 调试

源程序中难免会存在错误,错误一般可分为四类:

(1)编译错误:程序不符合 C 语言语法规则,在编译时将出错,编译错误包括语法错误(error)和警告错误(warning)。例如某一变量未定义先使用,则会出现语法错误。又如某变量未赋初值就用来求和,则会出现警告错误。

(2)逻辑错误:一个程序在编译时没有出现错误,执行后仍然得不到正确结果,这是由于在算法的设计过程或程序的表达式中存在错误,如表达式书写错误、程序控制流程错误等。

(3)运行错误:程序执行时在某些特殊情况发生的错误,如变量越界、除零错误等。

(4)连接错误:把目标程序连接成可执行程序时出现错误,如找不到库文件错误等。

程序调试是指对程序进行查错和排错。最常见的错误是编译错误和逻辑错误,有关程序的调试我们将在第十一章中详细叙述。

上述几个步骤在 Turbo C 2.0 集成开发环境中可以很方便地实现。上机操作的整个过程如图 1.1 所示:

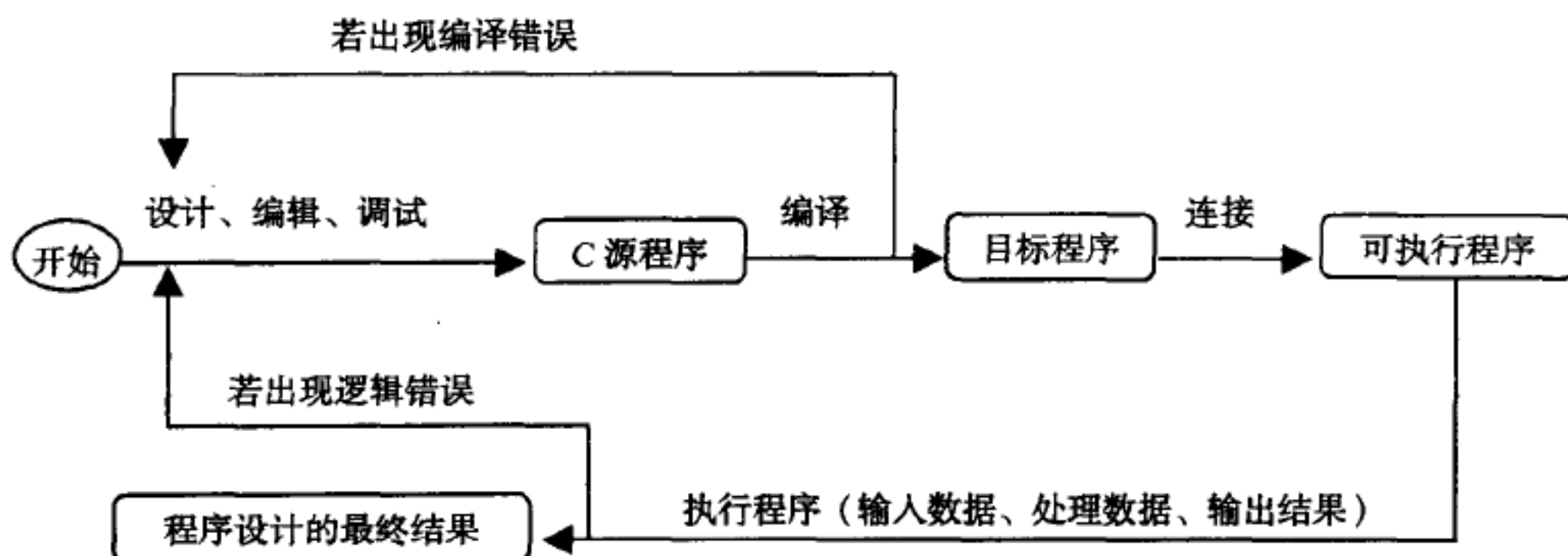


图 1.1 C 语言上机操作过程

1.3.2 Turbo C 的上机步骤

利用 Turbo C 2.0 集成开发环境可以非常方便地完成程序的编辑、调试、编译、连接和运行。以下通过一个简单的例子来说明 Turbo C 2.0 集成开发环境的上机步骤：

1. 在 DOS 状态下直接键入 tc 调用 Turbo C 程序。此时屏幕显示如图 1.2 所示的 Turbo C 主屏幕，按 **[Esc]** 键光标进入编辑窗口，这样就可以编辑源程序了。

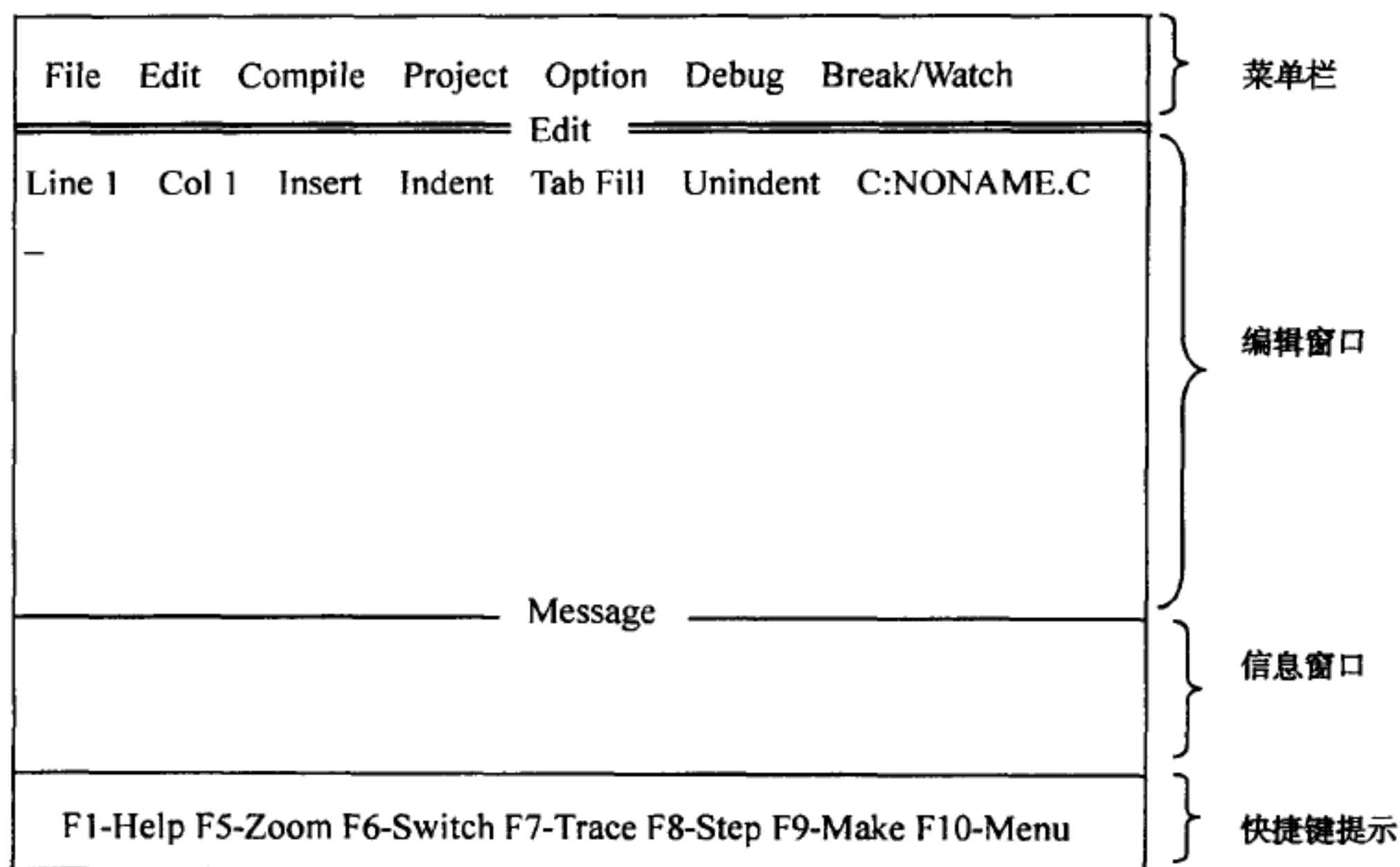


图 1.2 Turbo C 2.0 集成开发环境的主屏幕

2. [例 1.3] 编辑以下源程序：

```
#include "stdio.h"
main()
{
    float x, y;
    x = 25;
    y = x * x;
    printf("%6.2f 的平方是:%6.2f\n", x, y);
}
```

3. 按[F2]键文件存盘。若是第一次保存该文件,还应该再键入一个文件名,如 test,回车,这时上述源程序已经保存在磁盘上,其文件名为 TEST.C。若是再次保存该文件,则上次源程序的内容保存在 TEST.BAK 中,新修改后的源程序保存在 TEST.C 中。

4. 按[Ctrl]+[F9]完成源程序的编译、连接和运行。若发现错误,修改源程序后,重复步骤3和步骤4直至程序正确运行。养成运行程序之前先保存文件的好习惯,可以防止程序运行时死机而造成的源程序丢失。步骤4完成之后,将在磁盘上生成 TEST.OBJ 和 TEST.EXE 两个文件。

5. 按[Alt]+[F5]可以从 Turbo C 2.0 的主屏幕切换到用户屏幕,查看程序执行的结果。按任意键返回 Turbo C 2.0 集成开发环境。

[例 1.3]程序执行后结果是:

25.00 的平方是:625.00

6. 按[Alt]+[X]退出 Turbo C 集成开发环境,回到 DOS。我们可以看到磁盘上多了四个文件,它们分别是:TEST.C、TEST.BAK、TEST.OBJ 和 TEST.EXE。也可以在 DOS 环境中再次运行 TEST.EXE 文件。

综上所述,设计、编辑好一个 C 源程序后,只需要按[F2]、[Ctrl]+[F9]、[Alt]+[F5]三组键就可以完成程序的存盘、编译、连接、运行及查看结果。当然也可以利用菜单栏完成上述操作。有关程序的上机调试请参看第十一章。要熟练掌握 Turbo C 集成开发环境,请参阅附录 C“Turbo C 2.0 集成开发环境的使用”。

习 题

- 1.1 写出 C 语言的主要特点。
- 1.2 写出 C 语言程序的一般组成形式。
- 1.3 什么是程序设计? 什么是程序的编辑、调试、编译、连接及执行?
- 1.4 在 Turbo C 2.0 的集成开发环境中,怎样完成上机设计、编辑、调试、编译、连接及执行程序? 哪三组键就能完成程序的存盘、编译、连接、执行及显示程序执行结果?
- 1.5 在 Turbo C 2.0 的集成开发环境中完成例[1.1]及例[1.2]的上机操作。
- 1.6 参照本章例题,设计一个 C 程序,由键盘输入圆球的半径 R ,输出该圆球的表面积($4\pi R^2$)和体积($4\pi R^3/3$),其中 $\pi=3.14159$,并完成这个程序的上机操作。

第 2 章

C 语言的语法基础

程序设计语言都有自己的语法规则,必须严格遵循语法规则来编写程序,才能正确编译、连接、执行程序。本章叙述 C 语言的语法基础,包括标识符的命名规则、常量与变量的概念、基本输入/输出函数的用法、运算符与表达式的运算机制。

建议本章授课 10 学时,上机 4~6 学时,自学 12 学时。

2.1 基本数据类型

2.1.1 标识符与基本数据类型

1. 标识符

标识符(identifier)是一个名字,在 C 语言中标识符就是常量、变量、类型、语句、标号及函数的名称。程序设计语言中的标识符均有其命名规则。C 语言中标识符有三类:关键字、预定义标识符和用户定义标识符。

(1) 关键字

已被 C 系统所使用的标识符称为关键字,每个关键字在 C 程序中都有其特定的作用,关键字不能作为用户标识符。

以下是 Turbo C 2.0 的关键字,共 43 个,所有关键字中只包含小写字母和下划线。由 ANSI 标准定义的共 32 个关键字,见表 2.1。

表 2.1 ANSI 标准定义的共 32 个关键字

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

Turbo C 2.0 又增加了 11 个扩展的关键字,见表 2.2。

表 2.2 Turbo C 2.0 增加的 11 个扩展的关键字

asm	_cs	_ds	_es	_ss	cdecl
near	far	huge	interrupt	pascal	

(2) 预定义标识符

C语言系统提供的库函数名和编译预处理命令等构成了预定义标识符。在程序中若使用了库文件包含,就把相应的预定义标识符定义在程序中了,程序设计时就可以使用这些预定义标识符。有少数预定义标识符可以直接使用,而不用在程序中使用库文件包含。例如:在程序中不加#include "stdio.h"库文件包含,也可以调用printf、scanf这两个函数。如果程序中没有相应的库文件包含,用户可以定义与系统预定义标识符名称一样的标识符,但应尽量避免这样做。因为C语言系统已经规定了预定义标识符的特定含义,用户再定义与之相同的名字,便强行改变了系统原来赋予该标识符的意义,导致使用上的混淆。例如:若程序中没有#include "stdio.h"(相应的库文件包含),用户就可以定义putchar作为用户的函数名,但这与系统原有的预定义标识符putchar同名,调用该函数时,常常不清楚是调用系统的函数putchar还是调用用户定义的函数putchar。因此应尽量避免使用预定义标识符作为用户标识符。

(3) 用户标识符

用户可以根据需要对程序中用到的变量、符号常量、用户函数、标号等进行命名,成为用户标识符。在Turbo C 2.0中,用户标识符必须满足以下规则:

①标识符必须由英文字母、下划线、数字组成,不能包含其他字符(如全角字母和全角数字不能用于标识符);

②标识符必须由英文字母或下划线开头;

③标识符的长度不能超过32个字符;

④标识符大小写字母有区别(代表不同的标识符);

⑤标识符不能使用Turbo C 2.0的关键字。

用户在定义标识符时应注意以下事项:

①禁止使用Turbo C 2.0关键字作为用户的标识符。

②尽量避免使用预定义标识符作为用户标识符。

③标识符中不能出现全角字符、空格,不要把下划线“_”写成减号“-”。

④标识符必须先定义后使用,使用未经定义的标识符将出现编译错误。

⑤使用的标识符最好做到见名知义,以增加源程序的易读性和易维护性。例如area表示面积,sum表示求和等。

⑥在同一函数(的不同复合语句)中,最好不要定义相同的标识符作变量名。

表2.3举例说明了标识符的使用。

表 2.3 举例说明标识符的使用

正确的标识符	不正确的标识符	不正确的原因
area3	3area	数字打头
sort _ score	sort—score	标识符中使用了减号“-”
DEFAULT	default	使用关键字作标识符
a123b	a123b	标识符中使用了全角字符“3”
_915	-915	标识符中使用了减号“-”
xandy	x&y	标识符中含有非法字符“&”

2. 基本数据类型

程序是由处理对象和处理方法这两个主要要素组成的。处理方法指的是算法和程序设计

方法,而处理对象指的是数据结构。通常所说的“程序=数据结构+算法”就包含了这个意义。因此数据是程序的重要组成部分。数据有一个非常重要特征即数据的类型。数据类型不仅确定了变量的性质、取值范围、占内存空间大小,而且还确定了变量所能参加的各种运算方式。例如一个整型(int)类型的数据,在 IBM PC 系列微机上取值范围规定为 $-32768 \sim 32767$ 之间的整数,占内存空间 2 字节,能参与算术运算、位运算等。C 语言中,每个变量在使用之前必须定义其数据类型,每个常量也必须属于对应的数据类型。本节将介绍基本数据类型,构造数据类型分别在第四章和第五章中介绍。C 语言的数据类型如图 2.1 所示。

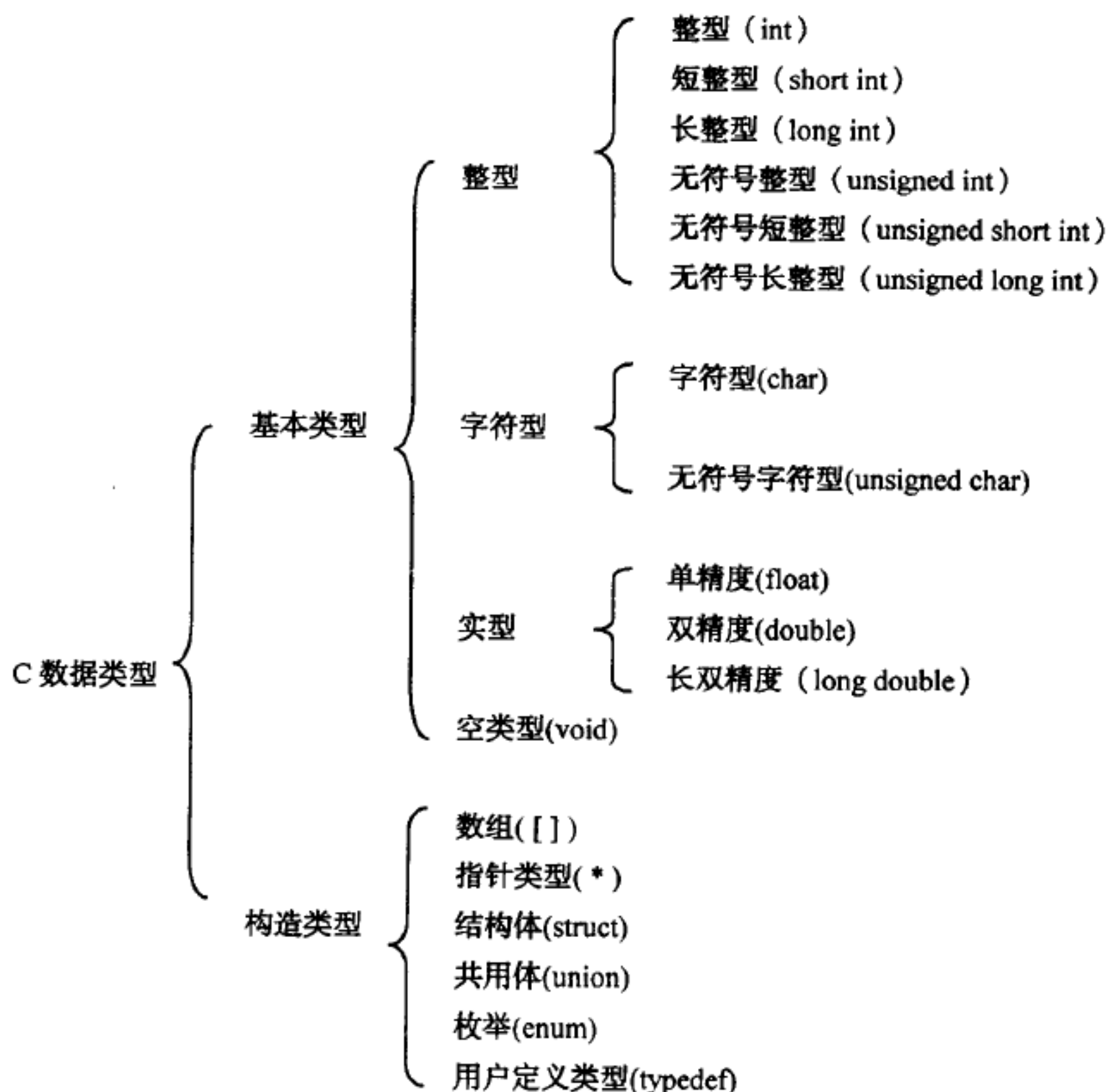


图 2.1 C 语言的数据类型

2.1.2 常量与变量

1. 常量

在程序执行过程中,值不能被改变的量称为常量。如 123、3.15、'A'、"Hello" 均是常量。在 C 语言中有整型常量、实型常量、字符型常量、字符串常量和符号常量五种类型。

(1) 整型常量

整型常量就是整数。C 语言的整型常量有三种表示形式:

①十进制整数。不由数字 0 开头的整数组成,可以由数字 0 至数字 9 组成,前面可加正号

“+”或负号“-”。如 123、+4560、-987 都是十进制整型常量。

②八进制整数。由数字 0 开头的整数组成,只能由数字 0 至数字 7 组成,前面可加正号“+”或负号“-”。例如:07623、-04567、+0315 都是八进制整型常量。如果写成 0891 则是错误的,八进制数不能含有数字 8 和数字 9。

③十六进制整数。由数字 0 和字母 x(或 X)开头的数组成,可以由数字 0 至数字 9、字母 a 至字母 f 或字母 A 至字母 F 组成,前面可加正号“+”或负号“-”。如 0xa3f、-0X9A、0x345、+0X6ab 都是十六进制整型常量。

使用整型常量应注意下面事项:

①一个整型常量的尾部加上字母 l 或 L 时,则为长整型(long 或 long int)常量。例如:123l 为十进制的长整型常量,0123L 为八进制的长整型常量,-0x123L 为十六进制的长整型常量。长整型常量往往用于函数调用中,如果函数的形参为长整型,则实参也必须是长整型。

②一个整型常量也可以由它的值确定它的类型,如果其值在 -32768~32767 范围内,则认为它是整型(int)常量;如果其值超出上述范围,而是在 -2147483648~2147483647 范围内,则认为它是长整型(long 或 long int)常量。

(2)实型常量

实型常量就是实数。C 语言中实型常量用两种形式表示:

①小数形式:一个实数的表示包括数字和小数点。例如 1.23456、-0.465、+789.123、0.0、1.0 等。

②指数记数法形式:这时实数包括整数部分、小数点、小数部分和指数部分,指数部分前加字母 e 或 E。例如 1.2345e3、12.345E2、1.2345e03、1.2345e+3 均表示 1234.5 这一实数。又如:0.123e+5、1e-4、35.69E11 均为合法的实数表示。用指数形式表示实数时,指数部分必须是整数,字母 e 或 E 之前必须有数字。例如 123e5.6、e5、.e9、e 等均为不合法的指数形式。

上述两种表示方法所表示的实型常量都是双精度实型(double),双精度实型常量在内存中占 8 个字节,取值范围在 $1.7 \times 10^{-308} \sim 1.7 \times 10^{+308}$ 之间。Turbo C 中,在双精度实型常量后加一字母 l 或 L 则构成长双精度实型常量(long double),长双精度实型在内存中占 10 个字节,取值范围在 $3.4 \times 10^{-4932} \sim 1.1 \times 10^{+4932}$ 之间。例如 1.23e1234l、5.67e-3456L 均为正确的长双精度实型常量。由实型常量的表示范围可知实型常量总是取正值。如果要使用负值,可在实型常量的前面加一个负号,构成常量表达式,其中的负号处理成算术操作符。实型常量只有十进制实型常量一种,没有八进制实型常量,也没有十六进制实型常量。绝对值小于 1 的实型常量,其小数点前面的零可以省略,如:0.123 可写为.123,-0.0123e-5 可写为-.0123e-5。

(3)字符型常量

C 语言的字符常量是 ASCII 码字符集里的一个字符,包括字母(大、小写有区别)、数字和标点符号以及特殊字符等,均为半角字符,一个字符常量在内存中占 1 个字节,因此字符常量不能是全角字符。C 语言字符常量有三种表示方法:

①把单个字符用一对单引号括起来表示字符常量。例如 'a'、'6'、'A'、'+'、':'。

②用该字符的 ASCII 码值表示的字符常量。例如十进制数 65 表示大写字母 'A',十六进制数 0x41 也表示 'A',八进制数 0101 还表示大写字母 'A'。一些不能用符号表示的特殊字符(如控制符等),可以用 ASCII 码值来表示,如换行可用 10 表示,也可用十六进制数 0x0a 或八进制数 012 来表示换行。

③反斜杠“\”开头后跟规定的单个字符或数字,并用一对单引号括起来表示字符常量。例

如用'\r'表示回车、用'\n'表示换行。换行也可用'\12'或'\012'反斜杠后跟八进制数表示,应注意这里反斜杠“\”后的八进制数、十六进制数前面的0可以省略,省略后并不表示成十进制数。换行还可用'\x0a'或'\0x0a'反斜杠后跟十六进制数表示。这种表示法中反斜杠“\”后面的字符变成了另外的意义,我们称之为转义字符。表 2.4 列出了常见的转义字符常量。

表 2.4 常见的转义字符常量

字符常量	含 义	等价表示
'\n'	输出到屏幕和文本文件为回车且换行,若输出到二进制文件仅为换行	10、0x0a、'\x0a'、'\12'
'\r'	回车	13、0x0d、'\x0d'、'\15'
'\t'	制表键,光标右移到下一输出区首,通常每个输出区占个 8 个字符	9、0x09、'\x09'、'\11'
'\f'	换页	12、0x0c、'\x0c'、'\14'
'\b'	退格	8、0x08、'\x08'、'\10'
'\\'	反斜杠字符 \	92、0x5c、'\x5c'、'\134'
'\''	单引号字符 '	39、0x27、'\x27'、'\47'
'\"'	双引号字符 "	34、0x22、'\x22'、'\42'
'\ddd'	1 到 3 位八进制数组成 ASCII 码所对应字符	0ddd
'\xhh'	1 到 2 位十六进制数组成 ASCII 码所对应字符	0xhh

[例 2.1]

```
main()
{
    printf("123456789012345\n");
    printf("ab c\tde\b\101fg\n");
    printf("ab c\rde\12\x41\n");
}
```

运行结果:

```
123 45678 9012345
ab c    dAfg
de c
A
```

上例没有单个输出字符常量,而是使用 printf 函数输出双引号内的各个字符。

第一个 printf 输出一串数字,作为坐标参照。最后的'\n'作用是回车并换行。

第二个 printf 从第 1 列开始先输出"ab c",后遇到'\t'光标移到下一输出区的开始位置(第 9 列),从第 9 列开始输出"de",又遇到退格符'\b',光标退格(左移一格)在字符'e'的位置上输出'\101' (字母'A')以及"fg",在第 10 列上字母'A'把字母'e'覆盖了,最后回车换行。

第三个 printf 从第 1 列开始先输出"ab c",后遇到'\r'回车,光标移到本行的第 1 列输出"de",并把"ab"覆盖了,又遇到回车换行'\12',光标移到下行的第 1 列输出字符'\x41' (字母'A'),最后回车换行。

(4) 字符串常量

若干个字符用双引号括起来就构成了字符串常量。如"Good morning!"、"123"、"A"、

"abcde"都是字符串常量。C语言中在存储字符串常量时,除了存储双引号中的所有字符之外,在字符串的最后还要存放一个字符'\0',表示该字符串常量到此结束。字符'\0'也称为字符串结束标志。因此字符串常量"abcde"占内存6个字节,字符串结束标志'\0'多占了一个字节。字符串常量"abcde"在内存是按如下方式存储的:

a	b	c	d	e	\0
---	---	---	---	---	----

利用C系统提供的输出函数printf和puts可以将字符串常量整体输出到屏幕,利用输入函数scanf和gets可以将字符串整体输入到内存。另外,字符常量'A'和字符串常量"A"不同,前者为字符常量,可直接赋值给字符变量,而后者为字符串常量,可以存放到字符数组中,也可以赋值给字符指针变量。'A'占内存1个字节,而"A"占内存2个字节,字符串结束标志'\0'多占了一个字节。

(5)符号常量

C语言中可以用一个标识符来代表一个常量,这个标识符就称为符号常量。可以用两种方式来定义C语言中的符号常量。

①利用宏定义#define来定义符号常量。例如:

```
#define PI 3.14159
#define ESC 27
#define ID "102343-3852396-y3v4x5a"
```

则PI、ESC与ID是符号常量,在程序中它们的值不能被改变。程序中用符号常量来代替一串不易记意的数字或一串字符串,不仅增加了程序的可读性,也减轻了程序设计人员的负担。另外,用一串较短的字符串来代替一串长字符串,也提高了程序的编写效率。C语言中习惯用大写字母表示符号常量。

②利用"const"来定义符号常量,这一方法在定义符号常量的同时也定义了该常量的数据类型。用const来定义符号常量的格式为:

```
const 数据类型关键字 符号常量1=常量1, 符号常量2=常量2,...
```

例如:

```
const int MAXINT=32767;
const long int MAXLONG=2147483647;
```

定义符号常量MAXINT为整型、MAXLONG为长整型。

2. 变量

在程序执行过程中,值可以改变的量称为变量。变量有以下几个特征:变量名、变量值、变量的数据类型、变量的地址、变量的存储类别、变量的作用域以及变量的生存期等。变量名是指按照C语言标识符的规则,给变量取的名称,使用变量名就可以存取变量的值。一个变量在内存中占有一定的存储空间,这个存储空间内所存放的数据就是变量的值。变量的数据类型确定了该变量的性质、取值范围、占内存空间大小以及所能参加运算的方式。有关变量地址的概念我们将在下一节讨论。在第六章讲述变量的存储类别、变量的作用域以及变量的生存期等。

在C语言中变量必须先定义才能使用。变量一经定义数据类型,计算机系统就会给该变量分配相应的存储空间,以便存放变量的值。一条变量定义语句由数据类型及其后所跟的一个或多个变量名组成。变量定义的格式如下:

```
数据类型关键字 <变量名表>;
```

变量名表是一个或多个标识符,每个标识符之间用逗号“,”分开。例如:

```
int i, j, k;
```

上述语句定义了三个变量,它们的名字分别为 i、j、k,数据类型为整型,它们可以存取整型数据,变量的取值范围是-32768~32767,占内存空间 2 个字节,可参与算术运算、位运算等。

表 2.5 给出了 Turbo C 2.0 基本数据类型关键字,还给出了在 IBM PC 系列微机上相应数据类型所对应的变量占存储空间大小以及变量的取值范围。

表 2.5 Turbo C 2.0 基本数据类型关键字、变量占内存字节数及变量取值范围

数据类型关键字	占内存字节数	取值范围	说 明
int	2	-32768~32767	整型
short int	2	-32768~32767	短整型
long int	4	-2147483648~2147483647	长整型
unsigned int	2	0~65535	无符号整型
unsigned short int	2	0~65535	无符号短整型
unsigned long int	4	0~4294967295	无符号长整型
char	1	-128~127	字符型
unsigned char	1	0~255	无符号字符型
float	4	$3.4 \times 10^{-38} \sim 3.4 \times 10^{38}$ 7~8 位有效数字	单精度实型
double	8	$1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$ 15~16 位有效数字	双精度实型
long double	10	$3.4 \times 10^{-4932} \sim 1.1 \times 10^{4932}$ 19~20 位有效数字	长双精度实型

表 2.5 中 long int 可简写为 long, short int 可简写为 short, unsigned int 可简写为 unsigned。表 2.5 列出的是 IBM PC 系列微机上基本数据类型所占内存字节数及数据的取值范围,对不同类型的机型,表中数据有所不同,使用时可查阅相关手册。本书的叙述、举例均以表 2.5 中的数据为标准。下面我们举例说明如何定义变量:

```
int i, sum;           定义 i、sum 为整型变量
long k, suml;         定义 k、suml 为长整型变量
double x, y;          定义 x、y 为双精度实型变量
float x1, x2;         定义 x1、x2 为单精度实型变量
char ch;              定义 ch 为字符型变量
unsigned u;           定义 u 为无符号整型变量
```

3. 变量的初始化

定义变量时,在变量之后加“=常量”,则对该变量进行了初始化。变量初始化过程是在定义变量类型时,把“=”号左边的常量赋值给该变量。例如:

```
int i=0, j=0;
```

上述语句定义 i、j 为整型变量,并把常量 0 赋给变量 i 与变量 j,这一初始化过程是在程序执行到本函数时给变量赋初值的。除了第六章讲述的静态存储变量和外部变量的初始化是在编译阶段完成的之外,动态变量的初始化过程都是在执行时完成。因此,语句 int i=0, j=0; 等价于以下两条语句:

```
int i, j;
```

```
i=j=0; /* 运行时赋初值,把0值赋值给变量i和变量j */
```

注意,若写成 `int i=j=0;`,则是非法语句。

一个动态变量未经初始化,也未对其赋值,其初值是不确定的。例如在某函数中有以下程序段:

```
int i;
```

```
printf("%d\n", i);
```

执行后输出结果是个不确定的整数值。

静态存储变量和外部变量若未经初始化,也对其未赋值,其初值是确定的,通常是0值,详见第六章。

2.1.3 内存的概念

计算机内存是由一片连续的存储单元组成,操作系统给每个单元一个编号,这个编号称为内存单元的地址(简称地址)。地址(编号)通常由一组连续的整数组成,编号小的称内存低地址,编号大的称内存高地址。每个单元占1个字节(byte)大小,这样内存中每一个字节就有一个地址(编号)。计算机在执行程序时先要做一系列的工作,例如要把程序的机器指令、常量等装入内存,在内存中为程序的变量分配存储空间等等,然后才完成程序预定的任务。

常量、变量在内存的存储情况经常用如图2.2(a)和图2.2(b)的形式表示,设在某程序中有以下定义变量的语句:

```
int i=25, j=0x1af, k=-25;
```

```
long int s=-25;
```

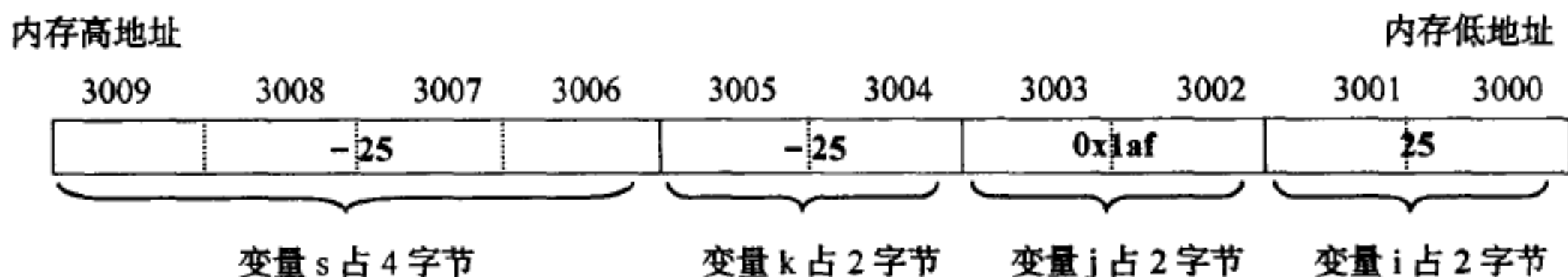


图 2.2(a) 变量在内存中分配的存储空间

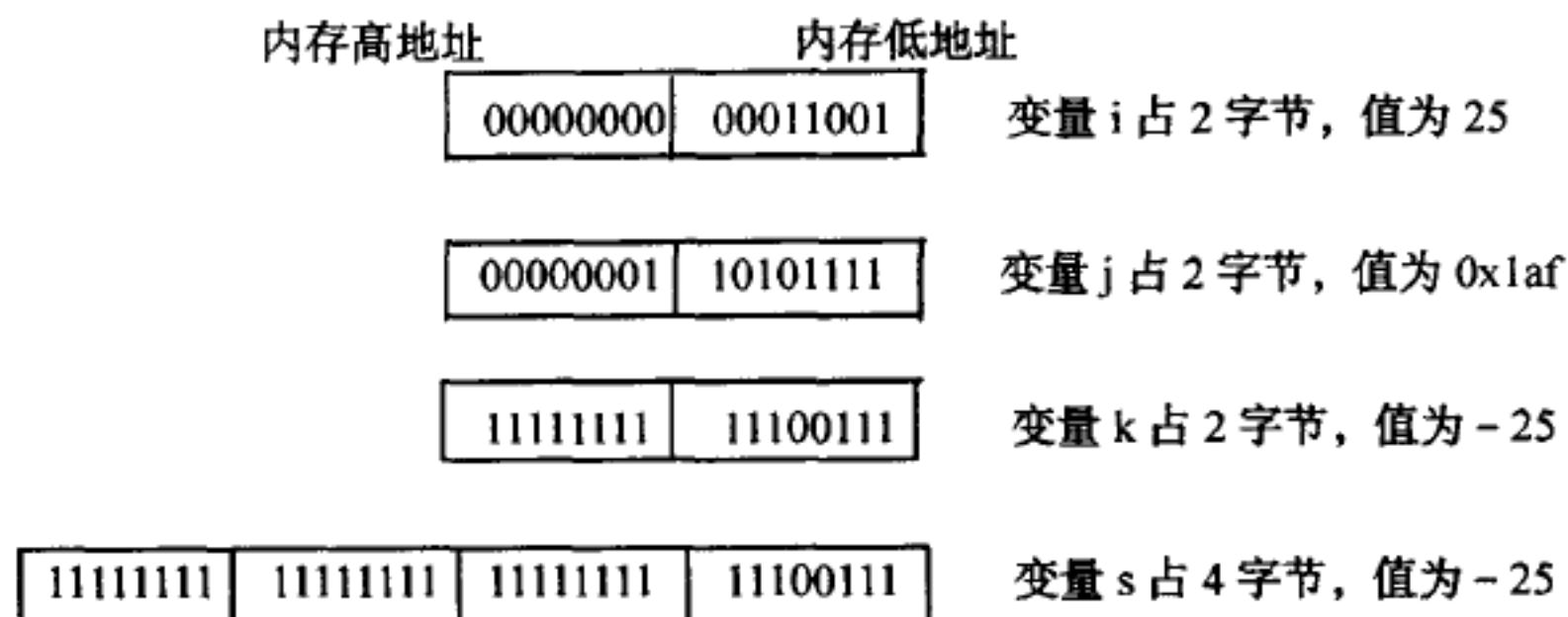


图 2.2(b) 变量的值在内存中按位表示

系统将在内存中为变量i、j、k各分配2个字节的连续存储单元,为变量s分配4个字节的

连续存储单元,并把相应的常量存储到该变量的地址所指的空间中。

设变量 i 分配到地址为 3000、3001 的 2 个连续存储单元,则这两个单元将存放常量 25;

设变量 j 分配到地址为 3002、3003 的 2 个连续存储单元,则这两个单元将存放常量 0x1af;

设变量 k 分配到地址为 3004、3005 的 2 个连续存储单元,则这两个单元将存放常量 -25;

设变量 s 分配到地址为 3006 到 3009 的 4 个连续存储单元,则这四个单元将存放常量 -25。

图 2.2(a)描述了变量 i 、 j 、 k 、 s 在内存中存储空间分配情况。

1 个字节占二进制 8 位(bit),有符号整数在内存中采用补码表示方法(详见附录 B),有符号数的最高位表示符号位,符号位为 0 表示正数,符号位为 1 表示负数。补码表示方法中正数用二进制原码表示,负数则用补码表示。上述语句中整型 `int` 类型 25 的二进制表示为 0000 0000 0001 1001, -25 的补码(按位取反,末位加 1)是:1111 1111 1110 0111。长整型 `long int` 类型 -25 的二进制补码是 1111 1111 1111 1111 1111 1111 1110 0111。图 2.2(b)描述了变量 i 、 j 、 k 、 s 的值在内存中按位表示情况。

无符号整数表示的都是正数,其最高位不是符号位,而是数据本身的一部分。假设内存中有一个二进制数据 1111 1111 1110 0111,我们把它视为有符号整数 `int` 类型,则值为 -25,若把它视为有无符号整数 `unsigned` 类型,则值为十六进制数 0xffe7(即十进制数 65511)。因此,内存中同一个数据,若将其视为不同的数据类型,其表现形式也不相同。

将一个字符型常量赋值给字符变量,并不把字符本身存到内存单元中,而是将该字符常量的 ASCII 码存储到内存单元中。例如字母 'A' 的 ASCII 码是 0x41(十进制数 65),若有语句 `char ch='A';`,则变量 ch 在内存中按位的表示如图 2.3 所示。

01000001

图 2.3 字符变量 ch 在内存中的按位表示, ch 占内存 1 字节

假设内存中有一个二进制数据 01000001,我们把它视为字符类型,则值为 'A',若把它视为 `int` 类型,则值为 65。例如在输出该数据时用 `printf("%c,%d\n",ch,ch);`,则输出 A,65。格式输出函数 `printf` 将在下一节介绍。

2.2 基本输入、输出函数

程序中通常应包含输入数据、处理数据和输出结果三个基本要素(见[例 1.1]与[例 1.2])。对于微型计算机来说,常见的输入设备有键盘、鼠标、扫描仪、数字化仪等,常见的输出设备有显示器、打印机、绘图仪等。程序设计中可以从磁盘文件输入数据,也可以将数据输出到磁盘文件,因此磁盘文件是既可输入也可输出的设备。在学习程序设计的入门阶段,先学习由键盘输入数据(通常指定键盘为标准输入设备),并把数据输出到屏幕(通常指定屏幕为标准输出设备)。在第八章中我们再介绍读写磁盘文件中的数据。C 语言没有提供输入、输出操作的语句。C 语言程序中的输入和输出完全依靠调用 C 语言的标准输入、输出函数来完成。Turbo C 2.0 库函数提供了格式化输入、输出函数和非格式化输入、输出函数。

2.2.1 格式输入函数和格式输出函数

C 语言库函数提供了针对标准设备的格式输入函数 `scanf` 和格式输出函数 `printf`。下面详细介绍这两个函数的用法。

1. printf 函数

printf 函数是格式化输出函数,用于向标准输出设备(通常指定为屏幕)按规定格式输出数据。printf 函数的调用格式为:

printf(格式化字符串, 输出表列);

其中格式化字符串包括两部分内容:普通字符与转义字符将按原样输出到屏幕;另一部分是“输出格式说明”,以“%”开始,后跟一个或几个格式字符,用来指定输出数据的格式。输出表列是若干个需要输出的数据项,称为函数的参数,可以是常量、变量或表达式。各参数之间用“,”分开。输出数据项与前面的“输出格式说明”必须由左至右一一对应,一个“输出格式说明”对应一个输出数据项。例如有如下程序段:

```
int i=1, j=255;
printf("i=%d,j=%0x\n", i, j);
```

格式化字符串

输出表列

上述程序段运行后结果如下:

i=1,j=ff

此处格式化字符串中的普通字符'i'、'='、','、'j'以及转义字符'\n'(回车换行)均原样输出到屏幕,数据项的输出格式分析如图 2.4 所示。

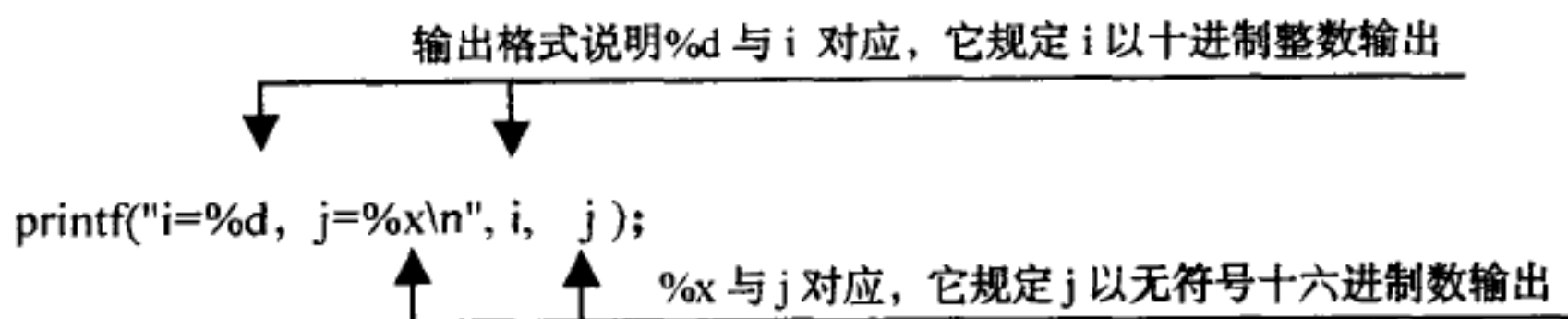


图 2.4 printf 函数的输出格式说明与输出项的对应关系

2. 调用 printf 函数时应注意的事项

(1) 格式化字符串中, 格式说明与输出项从左到右的数据类型必须一一匹配, 否则将输出错误结果。例如执行语句 `printf("%d,%d\n", 123, 123.456);`, 第一项 123 可以正确输出, 第二项将输出错误结果。这是因为第二个格式说明 `%d` 要求与之对应的输出项应是整型数据, 但此时输出项是实型数据, 与之不匹配, 产生输出错误。

(2)在格式化字符串中,格式说明与输出项的个数必须相同。如果格式说明的个数少于输出项的个数,多余的输出项不予输出。如果格式说明的个数多于输出项的个数,则对于多余的格式将输出不确定值。

(3)printf 在调用结束后将返回一个函数值,其值等于输出数据项的个数。

3. printf 函数的输出格式说明

每个格式说明都必须用%开头,以一个格式字符作为结束,在此之间根据需要可以插入

“宽度说明”、左对齐符号“-”、长度修饰符“l”和“L”等。

(1) 格式字符

格式字符用于规定输出不同的数据类型,格式字符和它们的作用如表 2.6 所示。

表 2.6 Turbo C 2.0 提供的输出格式字符及其作用

格式字符	作 用
d 或 i	输出十进制有符号整数(正数不输出符号“+”)
u	输出十进制无符号整数
o	无符号整数以八进制整数输出(输出时不带前导 0)
x 或 X	无符号整数以十六进制整数输出(输出时不带前导 0x 或 0X);x 用于输出 abcdef,X 用于输出 ABCDEF
c	以字符形式输出单个字符
s	输出字符串直至字符串结束标志'\0'为止,'\0'不输出
f	以小数形式输出实型数据,系统默认整数部分全部输出,小数部分输出 6 位小数,超长小数部分自动四舍五入
e 或 E	以指数形式输出实型数据,系统默认输出 1 位整数和 5 位小数,超长部分自动四舍五入,输出格式为:[-]m.dddde±dd,其中:m 为 1~9,d 为 0~9
g 或 G	由系统来选择%f或%e输出格式,输出 6 位有效数字,不输出小数尾部的 0
p	输出变量的内存地址
%	输出一个%号

(2) 长度修饰符

长度修饰符加在%和格式字符之间,对于长整型数一定要加 l。例如%ld 表示输出一个十进制长整型数据项。长度修饰符及其作用见表 2.7。

表 2.7 Turbo C 2.0 提供的长度修饰符及其作用

长度修饰符	作 用
F	数据项是 far 指针时使用
N	数据项是 near 指针时使用
l	格式字符是 d、i、o、u、x、X 时,用于输出长整型数据(long int)
L	格式字符是 e、E、f、g、G 时,用于输出长双精度实型数据(long double)

从上表可知:双精度实型数据的输出不必用%lf,只要用%f即可。但长双精度实型数据的输出必须加长度修饰符 L,例如%Lf、%Le 等。

(3) 输出数据所占的宽度

输出数据的宽度,可以使用系统默认宽度,也可以指定输出数据的宽度,主要有以下几种方式:

① 系统默认宽度

%格式字符(例如%d、%c、%u、%f等),输出数据所占的宽度由系统决定(通常取数据本身的宽度,不加空格)。

② 整型数据的输出宽度

下面的 m、n 表示非负整数,m 是输出数据的宽度,n 是输出数据小数部分的宽度。

%md、%mu、%mo、%mx,按 m 宽度输出数据,不足 m 个位数,左补空格(数据右对齐)。

输出宽度 m 前加一个负号“-”则右补空格(数据左对齐)。当数据的实际宽度超过 m 位时按实际长度输出。表 2.8 例举了各类不同整型数据的输出,输出格式化字符串中的“***”以原样输出,仅作坐标参照。

表 2.8 例举整型数据的输出

输出语句	输出结果	说 明
<code>printf("%d\n",12345);</code>	12345	以数据十进制的自身宽度 5 输出
<code>printf("%10d***\n",12345);</code>	12345***	以宽度 10 输出,左补 5 个空格
<code>printf("%-10d***\n",12345);</code>	12345***	以宽度 10 输出,右补 5 个空格
<code>printf("%10d***\n",-12345);</code>	-12345***	以宽度 10 输出,左补 4 个空格
<code>printf("%3d\n",12345);</code>	12345	以数据自身的宽度 5 输出
<code>printf("%ld\n",1234567890);</code>	1234567890	以数据自身的宽度 10 输出长整型
<code>printf("%d\n",1234567890);</code>	722	长整型数据,未加长度修饰符 l,输出错误
<code>printf("%u\n",12345);</code>	12345	以无符号十进制方式输出
<code>printf("%x\n",15915);</code>	3e2b	以数据十六进制的自身宽度 4 输出
<code>printf("%10X*****\n",15915);</code>	3E2B*****	以数据的十六进制输出,左补 6 个空格
<code>printf("%-15lo***\n",123456789);</code>	726746425***	以长整型数据的八进制输出,输出宽度为 15,数据自身宽度为 9 右补 6 个空格

③实型数据的输出宽度

`%m.nf`、`%m.ne`、`%m.ng` 数据输出的总宽占 m 列,其中小数部分占 n 列,若数据自身宽度小于 m 列,则左补空格。有关小数部分输出格式的规定如下:当输出数据的小数位多于指定的小数宽度 n 时,截去右边多余的小数,并对截去的第一位小数做四舍五入处理;当输出数据的小数位少于指定的小数宽度 n 时,在小数的最右边补 0。当输出数据的宽度大于指定的总宽度 m 时,小数部分仍按上述规则处理,整数部分原样输出。在输出宽度 m 前加一个负号“-”,若数据自身宽度小于 m 列,则右补空格。

应该注意并非所有输出的数字均是有效数字,单精度(float)7~8 位有效,双精度(double)15~16 位有效,长双精度(long double)19~20 位有效。格式说明中规定的数字宽度 m 和小数位宽度 n 再大也不能改变数据的存储精度,所输出的多余位的数字是无意义的。假设已有如下定义变量语句:

```
float f=12345.678;
```

```
double d=12345.6789056789;
```

```
long double ld=12345.6789056789e600l;
```

表 2.9 举例说明了使用 `printf` 函数输出上述变量 `f`、`d`、`ld` 各类实型数据。

④字符串的输出宽度

`%ms` 输出宽度占 m 列,若字符串宽度小于 m 列,左补空格,若字符串宽度大于 m 列,则原样输出字符串。

`%m.ns` 输出宽度占 m 列,但只取字符串的左端 n 个字符, n 小于 m ,左补空格。

在输出宽度 m 前加一个负号“-”,若字符串宽度小于 m 列,则右补空格。

表 2.10 举例说明字符串型数据的输出。

⑤字符数据的输出宽度

`%mc` 输出宽度占 m 列,左补 $m-1$ 个空格。在输出宽度 m 前加一个负号“-”,则右补 $m-1$ 个空格。表 2.11 举例说明字符型数据的输出。

(4)在输出的数字前加“+”号

使用%+格式字符(如%+d、%+f等),可在输出的数字前加“+”号。例如:

```
printf("%+d, %+d, %+10.2f", 123, -456, 12345.678);
```

运行结果:

```
+123, -456, +12345.68
```

表 2.9 例举实型数据的输出(变量 f,d,ld 在上面已经定义)

输出语句	输出结果	说 明
printf("%f\n",f);	12345.677734	以系统默认宽度输出单精度实数,整数部分原样输出,并输出 6 位小数,超出精度范围的数字无意义
printf("%f\n",d);	12345.678906	以系统默认宽度输出双精度实数,整数部分原样输出,并输出 6 位小数,小数后第 7 位四舍五入
printf("%10.2f\n",f);	12345.68	输出宽度为 10,其中整数部分原样输出,小数位占 2 位, 小数后第 3 位四舍五入,左补 2 个空格
printf("%-10.2f***\n",f);	12345.68 ***	输出宽度为 10,其中整数部分原样输出,小数位占 2 位, 小数后第 3 位四舍五入,右补 2 个空格
printf("%10.0f\n",f);	12346	不输出小位
printf("%10.5f\n",1.23);	1.23000	输出宽度为 10,其中整数部分原样输出,小数位占 5 位, 小数后右补 3 个 0
printf("%e\n",d);	1.23457e+04	以指数形式输出双精度实型数据,系统默认输出 1 位整数和 5 位小数,超长部分自动四舍五入
printf("%g\n",d);	12345.7	由系统来选择%f或%e输出格式,输出 6 位有效数字
printf("%Le\n",ld);	1.23457e+604	以指数形式输出长双精度实型数据,系统默认输出 1 位整数和 5 位小数,超长部分自动四舍五入

表 2.10 例举字符串型数据的输出

输出语句	输出结果	说 明
printf("%s\n","Hello!");	Hello!	以字符串实际宽度输出
printf("%10s\n","Hello!");	Hello!	输出宽度为 10,字符串为 6 个字符,左补 4 个空格
printf("%-10s***\n","Hello!");	Hello! ***	输出宽度为 10,字符串为 6 个字符,右补 4 个空格
printf("%10.3s\n","Hello!");	Hel	输出宽度为 10,取左边三个字符串 " Hel " 输出,左补 7 个空格
printf("%-10.3s***\n","Hello!");	Hel ***	输出宽度为 10,取左边三个字符串 " Hel " 输出,右补 7 个空格

表 2.11 例举字符型数据的输出

输出语句	输出结果	说 明
printf("%c\n",'A');	A	输出宽度为 1
printf("%5c\n",'A');	A	输出宽度为 5,左补 4 个空格
printf("%-5c***\n",'A');	A ***	输出宽度为 5,右补 4 个空格

(5)在输出数据前加前导 0

在指定输出宽度的同时,在数据前面的多余空格处填以数字 0。例如:


```
printf(" %010d, %010. 2f\n", -123, 12345. 678);
```

运行结果:

```
-000000123, 0012345. 68
```

(6)对输出的八进制数加前导0,对输出的十六进制数加前导0x

要在输出的八进制数加前导0,在输出的十六进制数加前导0x,可在%号和格式字符o和x之间插入一个#号。例如:

```
printf(" %#o, %#x\n", 65, 65);
```

运行结果:

```
0101, 0x41
```

4. scanf 函数

scanf 是格式化输入函数,可以从标准输入设备(通常指定为键盘)上,以各种不同的格式读入数据到变量。scanf 函数的格式为:

```
scanf(格式化字符串, 地址表列);
```

其中格式化字符串包括以下三类不同的字符:

- (1)输入格式说明,“输入格式说明”与 printf 函数中的“输出格式说明”基本相同;
- (2)空白字符,指空格键、回车键(Enter 键)和制表键(Tab 键);
- (3)非空白字符,C 语言的字符常量或字符串常量。

地址表列是若干个需要读入数据的地址项。各个地址项之间用“,”分开。地址项必须是 C 语言中合法的地址表达式,通常是变量的地址。此处的地址项称为输入项。例如有如下程序段:

```
int i;
float f;
scanf(" %d%f", &i, &f);
```

输入格式化字符串, 地址表列(输入项表列)

```
printf("i= %d, f= %8. 3f\n", i, f);
```

上述程序段运行后从键盘输入:123 456.789 回车

运行结果:

```
i=123, f= 456.789
```

此处“&”是“地址运算符”,&i 表示变量 i 在内存中的地址。scanf 函数先遇到输入格式说明 %d 先读入一个整型数 123,然后遇到输入格式说明 %f,略去输入中的一个或多个连续空白字符后,读入一个实型数 456.789,读入的数据存入相应的内存地址中,如图 2.5 所示:

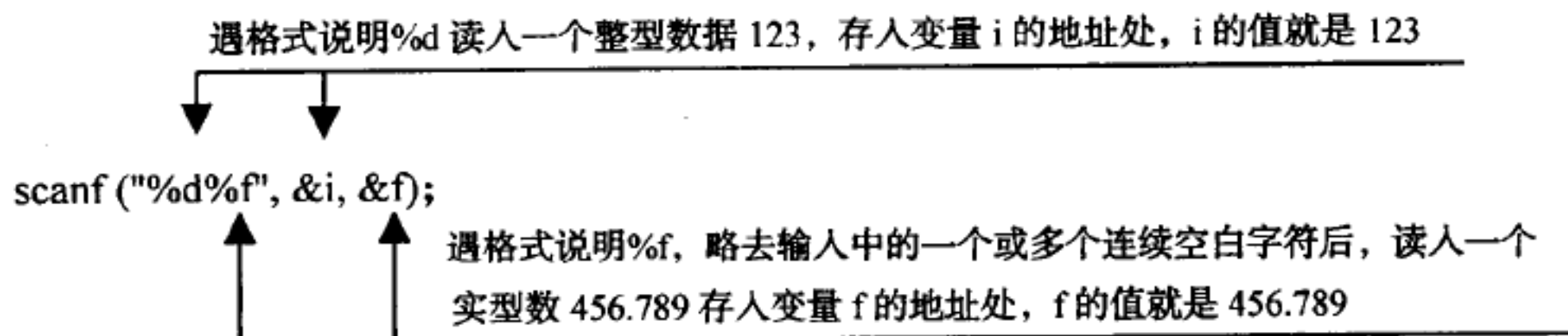


图 2.5 scanf 函数的输入格式说明与输入项的对应关系

如果输入的第一个数据前面有空白字符,则也要先略去这些空白字符,再开始读入第一个数据。例如上述程序段运行后从键盘输入:回车回车 123 456.789 回车
屏幕上输出结果不变:

```
i=123,f= 456.789
```

5. 调用 scanf 函数时应注意的事项

(1) 格式化字符串中,格式说明的类型必须与地址表列中输入项的类型由左至右一一对应匹配。如果类型不匹配,将不能读入正确数据。例如:

```
int i;
float f;
scanf("%d%d", &i, &f);
```

此处变量 i 可以正确读入数据,而变量 f 得不到正确数据。第二个格式说明 %d 要求与之对应的输入项是整型变量的地址,但 &f 为实型数据的地址与 %d 不匹配,因此出错。

(2) 当输入的数据少于输入项时,程序等待输入,直到所有输入项都读入数据为止,若有非法字符读入,scanf 函数也结束输入。如下程序段:

```
int i, j, k;
scanf("%d%d%d", &i, &j, &k);
```

当程序运行到上述 scanf 语句时,从键盘输入:

```
123 456 回车
```

此时把 123 赋值给变量 i,把 456 赋值给变量 j,由于还有一个输入项没有读入数据,程序等待输入,此时若再从键盘输入:

```
789 回车
```

把 789 赋值给变量 k,scanf 函数结束输入。

如果输入的数据多于输入项时,多余的数据存储在输入缓冲区,留作下一个输入项的输入数据。系统为标准输入设备(通常指定为键盘)在内存开辟了一块区域作为输入缓冲区。由键盘输入的数据先存储在输入缓冲区中,程序再从输入缓冲区读取数据,没取完的数据就暂存于输入缓冲区中,等待程序取走。

(3) 格式化字符串中,格式说明的个数应该与输入项的个数相同。

若格式说明的个数少于输入项的个数时,scanf 函数结束输入,多余的数据项没接收新的数据。例如下例程序段:

```
int i=10, j=20, k=30;
scanf("%d%d", &i, &j, &k);
```

执行时由键盘输入:1 2 3 回车

结果变量 i 的值为 1,j 的值为 2,但 &k 没有与之对应的格式说明,输入的数据 3 将不会赋给变量 k,变量 k 仍保持原来的值 30。多余的输入数据 3 存储在输入缓冲区,留作下一个 scanf 函数输入项的输入数据。

若格式说明的个数多于输入项的个数时,scanf 函数结束输入,每个多余的格式说明将滤掉输入缓冲区中的一个输入数据。例如有如下语句段:

```
int i, j, k;
scanf("%d%d%d", &i, &j);
scanf("%d", &k);
```

执行时由键盘输入:10 20 30 40 回车

结果把10赋给变量*i*, 20赋给变量*j*, 第一个scanf函数中多了一个格式说明"%d"将输入的30滤掉, 然后把40赋给变量*k*。系统将提示错误信息:Null pointer assignment(指定了空指针), 说明使用方法不正确。要滤掉输入缓冲区中的一个输入数据, 正确的方法是在%与格式字符之间加一个"*"号。例如把上面程序段的第一个scanf改成:

```
scanf("%d%d%*d", &i, &j);
```

%*d表示从输入缓冲区中读走一个数据后不赋给任何变量。这时再运行上述程序段读入数据时, 30被跳过去, 把40赋值给变量*k*。这样就不会出现错误信息了。

(4)从键盘输入数据时, 输入的各数据之间用空白字符(空格键、回车键Enter或制表键Tab)隔开。空白字符可以一个, 也可以连续几个。

(5)从键盘输入数据时, 按空格键或制表键后, 左边的数据并没有被程序接收, 需要最后按一下回车键, scanf函数才能接受从键盘输入的数据。

(6)如果在格式化字符串中插入有某个非空白字符, 输入数据时应输入一个与该非空白字符相同的字符, 形成一一对应。例如下面程序段:

```
int i=1, j=2, k=3;
scanf("%d,%d,%d", &i, &j, &k);
```

要求键盘输入的每个数据之间紧跟一个逗号, 下面是正确的输入:

123,456,789 回车

以下也是正确的输入:

123, 456, 789 回车

但是下面的输入不正确:

123 456 789 回车

以下的输入也不正确:

123; 456; 789 回车

因为在格式化字符串中有逗号",", 就要求输入一个","与之对应, 如果","这一字符没有找到, 本句scanf函数就结束输入, 后两种输入方法只能把123赋值给变量*i*, 变量*j*、*k*的值不变, *j*的值仍为2, *k*的值仍为3。再如, 执行下述语句时:

```
scanf("i=%d,j=%d,k=%d", &i, &j, &k)
```

正确的输入数据格式是:

i=123,j=456,k=789 回车

(7)如果在格式化字符串中插入若干个空白字符(空格键、回车键和制表键), 输入数据时只要输入一个(或多个连续空白字符)。例如有如下程序段:

```
int i,j;
scanf("%d _ _ _ %d", &i, &j);
```

输入时两个数据之间应输入一个(或多个)空白字符, 以下各行的输入均正确:

123 _ 456 回车

123 回车 456 回车

123 _ _ _ 456 回车

这三种输入都把123赋值给变量*i*, 456赋值给变量*j*。

(8)scanf 函数的地址表列中的输入项是地址,不是变量名,因此普通变量前应加“&”地址操作符。但是对于字符串数组或字符串指针变量(见第四章和第五章内容),其变量名本身就是地址,不需要在它们前面加“&”地址操作符。

(9)scanf 函数在调用结束后将返回一个函数值,其值等于有得到值的输入项的个数。

6. scanf 函数的输入格式说明

每个格式说明都必须用%开头,以一个格式字符作为结束,在此之间根据需要可以插入“宽度说明”、长度修饰符“l”和“L”等。

(1)格式字符

格式字符用于规定输入不同的数据类型,格式字符和它们的作用如表 2.12 所示。

表 2.12 Turbo C 2.0 提供的输入格式字符及其作用

格式字符	作 用
d	输入十进制整数
i	输入十进制整数,以前导 0 开始的八进制整数或 0x 开始的十六进制整数
o	输入八进制整数(可以带前导 0,也可以不带前导 0)
x	输入十六进制整数(可以带前导 0x 或 0X,也可以不带前导 0x 或 0X)
u	输入无符号十进制整数
c	以字符形式输入单个字符
s	输入字符串,遇第一个空白字符结束
f, e	以小数形式或指数形式输入实型数据

(2)长度修饰符

长度修饰符加在%和格式字符之间,长整型数一定要加 l。例如%ld 表示输入一个十进制长整型数据项;双精度实型数据的输入必须加长度修饰符 l,例如%lf、%le。长双精度实型数据的输入必须加长度修饰符 L,例如%Lf、%Le。长度修饰符及其作用见表 2.13

表 2.13 Turbo C 2.0 提供的长度修饰符及其作用

长度修饰符	作 用
l	格式字符是 d、i、o、u、x 时,用于输入长整型数据(long int 或 unsigned long)
l	格式字符是 f、e 时,用于输入双精度实型数据(double)
L	格式字符是 f、e 时,用于输入长双精度实型数据(long double)

(3)输入数据的宽度

scanf 函数输入数据的实际宽度是由输入数据的结束标志决定的,因为在读入某数据项时,遇到结束标志则完成该数据项读入。结束标志有三种:

①空白字符:空格键、回车键或制表键(Tab)。

②宽度 m:格式字符前可用一个整数 m 指定输入数据所占宽度,此时输入数据的宽度不能大于 m。在 scanf 函数中,不能指定实型数据小数位的宽度。

③非法字符:由于非法字符的存在,构成了不正确的 C 常量。例如在输入整数时输入 123r5,此处字母 r 为非法字符。

表 2.14 举例说明 scanf 函数的用法。以下的各例句中,i、j 为整型变量(int),k 为长整型变量(long),ch 为字符型变量(char),f 为单精度实型变量(float),d 为双精度实型变量(double),ld 为长双精度实型变量(long double)。

表 2.14 举例说明 scanf 函数的用法

语句: scanf("%3d%8f%d", &i, &f, &j);
键盘输入数据:123456.7891234 回车
结果:i 的值为 123, f 的值为 456.7891, j 的值为 234。
说明:遇到宽度 3 和 8,数据项结束。
语句: scanf("%ld%c%d", &k, &ch, &i);
键盘输入数据:123456.7890 回车
结果:k 的值为 123456, ch 的值为 ".", i 的值为 7890。
说明:遇到非法字符 ".",数据项结束。
语句: scanf("%ld%lf%Lf", &k, &d, &ld);
键盘输入数据:12345678 回车 回车 1234567.1234 回车 123.456e+789 回车
结果:k 的值为 12345678, d 的值为 1234567.1234, ld 的值为 1.23456e+791。
说明:遇到空白字符,数据项结束。由于 k 为长整型变量,d 为双精度实型变量,ld 为长双精度实型变量,此处长度修饰符 l 与 L 是必须的,不能省略,否则无法正确读入数据。

(4)关于格式说明%c

在 scanf 函数中的格式说明%c 用于输入单个字符,这时从键盘输入的空白字符将作为有效字符输入。例如:

```
char c1, c2, c3;
scanf("%c%c%c", &c1, &c2, &c3);
```

如果键盘输入:A _B _C

字符'A' 赋值给变量 c1,字符'_'赋值给变量 c2,字符'B' 赋值给变量 c3。使用格式说明%c 时空白字符不作为数据间的间隔,因此'_'作为下一个字符赋值给变量 c2。回车键和制表键遇%c也同样赋值给相应的字符变量。

(5)连续使用多个 scanf 函数

[例 2.2] 在程序中连续使用多个 scanf 函数时,应注意消除前一行输入的回车符。

```
#include "stdio.h"
main()
{
    int i, j;
    float x=0.0, y=0.0;
    char c1, c2;
    scanf("%d%d", &i, &j);
    scanf("x=%f y=%f", &x, &y);
    scanf("%c%c", &c1, &c2);
    printf("i=%d,j=%d\n", i, j);
    printf("x=%6.2f,y=%6.2f\n", x, y);
    printf("c1=%c,c2=%c\n", c1, c2);
}
```



```
}

```

程序运行时,要求从键盘把 1 赋给 i,2 赋给 j,1.1 赋给 x,2.2 赋给 y,'a' 赋给 c1,'b' 给 c2。

键盘输入:

```
1 2 回车

```

```
x=1.1 y=1.2 回车

```

运行结果:

```
i=1,j=2

```

```
x= 0.00,y= 0.00

```

```
c1=

```

```
,c2=x

```

上述结果显然不正确,这是因为第一行输入的“回车”被第二个 scanf 函数所接收,而与第二个 scanf 函数要求输入“x=…”不符,第二个 scanf 函数立即停止执行,转去执行第三个 scanf 函数,把第一行输入的“回车”赋值给 c1,再把 'x' 赋值给 c2。

解决方法一,在第二、第三个 scanf 的格式字符串前加一个“空格”以抵消上一行输入的“回车”,如下:

```
scanf(" %d%d", &i, &j);

```

```
scanf(" x= %f", &x, &y);

```

解决方法二,分别在第二、第三个 scanf 函数前加一条函数调用语句 fflush(stdin);。本语句的作用是清除当前标准输入缓冲区内存放的数据,这样无论上一行输入了什么内容都作废,下个 scanf 将接收键盘输入的新数据。如下:

```
fflush(stdin);

```

```
scanf("x= %f", &x, &y);

```

```
fflush(stdin);

```

```
scanf(" %c%c", &c1, &c2);

```

按上述方法更改后,执行[例 2.2]时,键盘输入:

```
1 2 回车

```

```
x=1.1 y=1.2 回车

```

```
ab 回车

```

运行结果:

```
i=1,j=2

```

```
x= 1.10,y= 2.20

```

```
c1=a,c2=b

```

2.2.2 非格式化输入、输出函数

标准格式化输入、输出函数可以指定输入、输出的格式,使用起来方便、如意,但函数编译后程序代码较大。某些程序不要求指定输入、输出格式,这时用非格式化输入、输出函数也非常方便,并且函数编译后程序代码较小,效率高。非格式化输入、输出函数主要用于字符和字符串的输入和输出,调用时要在程序中使用文件包含 #include "stdio.h"。本节介绍单个字符的非

格式化输入、输出函数,关于字符串的非格式化输入、输出函数我们将在第六章介绍。

1. putchar 函数

putchar 函数把一个字符输出到标准输出设备(通常指定为显示器)上。其调用格式为:

```
putchar(ch);
```

其中 ch 为一个字符变量或一个字符常量(包括转义字符常量),putchar(ch)函数与 printf("%c", ch)函数的作用等同。例如以下程序段每条语句均向屏幕输出字符“A”。

```
putchar('A');    putchar('\101');    putchar('\x41');
putchar(65);     putchar(0101);     putchar(0x41);
```

2. getchar 函数

getchar 函数的作用是从标准输入设备(通常指键盘)上读入一个字符。其调用形式为:

```
getchar();
```

它接收标准输入缓冲区中的一个字符,对空白字符也一并接收,并带回显(显示键盘所按的字符)。执行到 getchar 函数时,等待输入字符,直到按回车才结束,回车前的所有输入字符都将逐个显示在屏幕上。但只有第一个字符被 getchar 函数接收。

[例 2.3]

```
#include "stdio.h"
main ()
{
    char ch;
    ch=getchar();
    /* 从键盘读入字符直到回车结束,把第一个字符赋值给 ch */
    putchar(ch); /* 显示输入的第一个字符 */
    putchar('\n'); /* 回车换行 */
    getch(); /* 等待按任一键程序结束 */
}
```

3. getch 和 getche 函数

getch 和 getche 函数的作用都是从标准输入设备(通常指键盘)上读入一个字符。其调用形式分别为:

```
getch();
getche();
```

getch 函数读入的字符不会回显在屏幕上,而 getche 函数将读入的字符回显到屏幕上。它们与 getchar 函数不同之处在于:getch 和 getche 函数不必输入回车才结束,只要输入一个字符,该字符立即被接收,程序继续执行下一条语句。利用这一特点,这两个函数经常用于交互输入的过程中完成暂停功能。

[例 2.4] 输入一位学生三门课学习成绩,求出该学生三门课平均成绩,按 Esc 退出。

```
#include "stdio.h"
const char ESC=27;
main()
{
    float f1, f2, f3, aver;
```

```

do    /* do/while 循环语句条件成立时,反复执行{...}中的语句 */
{
    printf("请输入学生的三门课学习成绩:");
    scanf("%f%f%f", &f1, &f2, &f3);
    aver=(f1+f2+f3)/3.0;
    printf("该学生的三门课平均成绩是:%6.2f\n", aver);
    printf("按 Esc 退出,按其他键继续...\n");
} while( getch()!=ESC ); /* 等待,按 Esc 退出,按其他键继续计算 */
}

```

运行结果:

请输入学生的三门课学习成绩:67.5 78.5 89

该学生的三门课平均成绩是: 78.33

按 Esc 退出,按其他键继续...

按其他键继续计算,直到按下 Esc 键,程序结束。

2.3 运算符与表达式

C 语言有很丰富的运算符,总结如下:

1. 算术运算符: + (加), - (减), * (乘), / (除), % (求余)
2. 关系运算符: > (大于), < (小于), <= (小于等于),
 >= (大于等于), == (等于), != (不等)
3. 逻辑运算符: ! (逻辑非), && (逻辑与), || (逻辑或)
4. 位运算符: ~ (按位取反), & (位与), | (位或),
 ^ (异或), << (左移), >> (右移)
5. 条件运算符: ? :
6. 指向结构体成员运算符: ->
7. 结构体成员运算符(分量运算符): .
8. 自增1,自减1运算符: ++, --
9. 类型转换运算符: (类型)
10. 指针,取地址运算符: * (指针运算符), & (取地址运算符)
11. 下标运算符: []
12. 赋值运算符: =, *=, +=, -=, /=, %=, >>=, <<=, &=, ^=, |=
13. 逗号运算符: ,
14. 长度运算符: sizeof()
15. 负号运算符: -
16. 括号运算符: ()

本章只介绍算术运算符、增1与减1运算符、关系、逻辑及条件运算符、位运算符、赋值运算符、类型转换运算符、逗号运算符和长度运算符。有关运算符的优先级和结合性的小结请参见附录 E。

2.3.1 算术运算

1. 算术运算符

C语言的算术运算符有以下5种:

+ (双目运算两数相加,如 $5+6$;单目运算取正值,如 $+15$)

- (双目运算两数相减,如 $5-6$;单目运算取负值,如 -19)

* (双目运算符,两数相乘,如 $5*6$)

/ (双目运算符,两数相除,如 $15/6$)

% (双目运算符,取模或求余数,两个数必须都是整数,如 $15\%6$ 值为3)

所谓单目运算符是指对一个运算对象进行操作,例如 -19 。双目运算符是对两个运算对象进行操作,这两个运算对象分别放在操作符的左边和右边,如 $5+6$ 。

2. 算术运算符的优先级和结合方向

计算机语言中的运算符与数学中的运算符类似,都有优先级和结合方向。C语言的算术运算符的优先级如下(同一行上的运算符,优先级相同):

()	圆括号	高
+, -	单目运算符,取正、取负	↓
*, /, %	双目运算符,乘、除、取模	
+, -	双目运算符,加、减	
		低

上面所有双目算术运算符的结合方向都是“从左到右”,而单目运算符取正“+”和取负“-”的结合方向是“从右到左”。

3. 算术运算表达式

C语言的算术表达式是由算术运算符把运算对象连接起来,构成合法的式子。运算对象包括常量、变量和函数。算术表达式的值为整数或实数,如 $3*x+1.0/y-10*\text{sqrt}(x)$ 。

在对算术表达式进行运算时,应注意以下几点:

(1)算术表达式中可以使用多层圆括号,左、右括号必须配对。运算时先计算出内层括号表达式的值,由内向外计算表达式的值。

(2)取模运算符%两侧的运算对象必须是整数,运算结果是两数相除后所得的余数。在多数机器上,取模后值的符号与运算符左侧运算对象的符号相同。例如: $5\%3$ 值为2, $-5\%3$ 值为-2, $5\%(-3)$ 值为2;实数不能参与取模运算,如 $5\%1.5$ 是非法的算术表达式。

(3)整数除:两个整数相除后值等于商的整数部分(小数部分没有四舍五入),例如 $1/2$ 为整数除,其值为0。

(4)实数除:两个相除的数中至少有一个是实数,相除后的值等于(商本身)实数,如 $1.0/2$ (或 $1/2.0$ 或 $1.0/2.0$) 均为实数除,值都是0.5。

例如表达式 $(-16/3*2+1)\%6$ 的值是-3,先计算圆括号内的值,单目运算符“-”优先级高于其他双目运算符,先计算整数除 $-16/3$ 值为-5,然后 $-5*2+1$ 值为-9,最后 $-9\%6$ 值为-3。

4. 各类数值型数据的混合运算

在C语言中允许整型、实型、字符型数据进行混合运算。例如:

1. $23+'A'+456\%'B'$

是合法的 C 表达式。不同类型的数值型数据进行混合运算时,先要把低数据类型向高数据类型转换,转换为同一类型后才进行运算。转换的规则如图 2.6 所示。

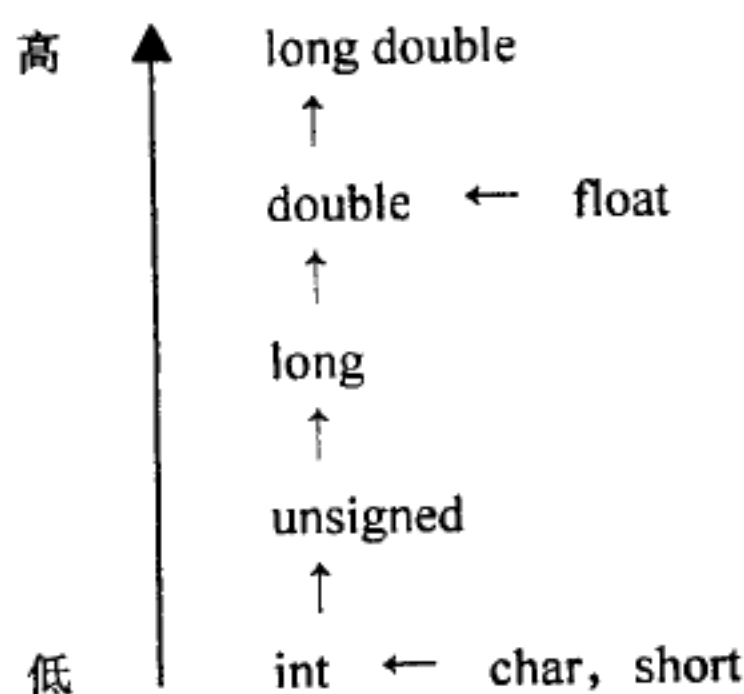


图 2.6 不同类型的数值型数据进行混合运算时类型的转换规则

图中向左的横向箭头表示必须进行的转换,如两个 float 型的数据相加,也要先把这两个 float 数据转换成 double 型数据,然后再进行运算,以提高精度。向上的纵向箭头表示不同类型数据混合运算时,先要进行的数据类型转换。例如表达式 $123.456 + 543 - 'A'$,运算时先把整型数据 543 转换成 double 型数据,与 123.456 相加值为 666.456 (double 型数据),然后把字符 'A' 转换成 65.0 (double 型数据),再进行相减运算,最后结果为 601.456 (double 型数据)。

2.3.2 增1与减1运算

增1运算符“++”使运算对象的值增1,而减1运算符“--”则使运算对象的值减1。它们都是单目运算符,其运算对象必须是变量,不能是常量和表达式。例如语句 $i++$; 相当于语句 $i = i + 1$; ,又如语句 $i--$; 相当于语句 $i = i - 1$;

增1与减1运算符可以作前缀运算符,这时是先使运算对象值增1或减1之后,再使用运算对象。例如 $i = 1; j = ++i$; ,则变量 i 的值先增1变成2,然后把 i 的值赋给变量 j, j 的值为2。同样地,若有语句 $i = 1; j = --i$; ,执行后 i 的值为0, j 的值为0。

增1与减1运算符也可以作后缀运算符,这时是先使用运算对象,再使运算对象值增1或减1。例如 $i = 1; j = i++$; ,则先把 i 的值赋给变量 j, j 的值为1,然后变量 i 的值再增1变成2。同样地,若有语句 $i = 1; j = i--$; ,执行后 i 的值为0, j 的值为1。

增1与减1运算符如果仅仅只进行自加、自减运算,没有使用运算对象的值,作前缀运算符与作后缀运算符运算结果一样。如语句 $i--$; 与语句 $--i$; 效果一样,都等同于语句 $i = i - 1$;

“++”和“--”运算符的结合方向是“从右到左”。例如 $i = 1; j = -i++$; ,由于取负运算符“-”和增1运算符“++”的优先级相同,结合方向是“从右到左”,即相当于 $-(i++)$,又由于是后缀运算符,则先取出 i 的值使用,把 -i 的值赋值给 j, 变量 j 的值为 -1,然后使 i 的值增1变成2。

应尽量避免在一个表达式中多次用“++”和“--”运算符,这样写出的表达式可读性差,不同的编译系统也可能给出不同的运算结果,例如 $i = 1; j = ++i + i++$;

$k=i++*++i$ 。可使用临时变量过渡一下,分解成多条语句来完成类似的功能。例如上面三条语句可以改成如下五条语句:

```
i=1;
t=++i;      /* 执行后,i 值为2, t 值为2 */
j=t+i++;    /* 执行后,j 值为4, i 值为3 */
t=++i;      /* 执行后,i 值为4, t 值为4 */
k=(i++)*t;  /* 执行后,k 值为16,i 值为5 */
```

变量 t 作为临时变量,把“++”运算符只写在一个表达式中,这样程序可读性好,不同的编译系统给出的运算结果都相同。以上程序段运行后,变量 i 、 j 、 k 的值分别是5、4、16。

2.3.3 关系、逻辑及条件运算

1. 关系运算

(1) 关系运算符

关系运算符用于比较两个运算对象的大小。C语言提供六种关系运算符:

<(小于), >(大于), <=(小于或等于), >=(大于或等于)

这四个运算符优先级相同,但都高于下面两种关系运算符:

==(等于), !=(不等于)

这两个运算符优先级也相同,都低于上面四种关系运算符。关系运算符是双目运算符,结合方向是“从左到右”。关系运算符的优先级低于算术运算符,但高于赋值运算符。

(2) 关系表达式

用关系运算符把C的合法表达式联系起来就构成了关系表达式。这里的合法表达式可以是算术表达式、关系表达式、逻辑表达式、赋值表达式、逗号表达式、字符表达式等。通常关系表达式的值为一个逻辑值:“真”或“假”。C语言中没有专门用来表示“真”、“假”的常量,在C语言中的非零值被认为是“真”,零则被认为是“假”。关系表达式的值只能是1或0。关系表达式值为1,称该关系表达式为“真”;若关系表达式值为0,则称该关系表达式为“假”。例如:

$123 >= 456$	值为0
$1 > (123 > 456)$	值为1
$'A' < 'a'$	值为1

2. 逻辑运算

(1) 逻辑运算符

C语言提供三种逻辑运算符:&&(逻辑与),||(逻辑或),!(逻辑非)。其中“!”是单目运算符,“&&”和“||”是双目运算符。逻辑运算符的优先级如图2.7。

“!” → 算术运算符 → 关系运算符 → “&&” → “||” → 赋值运算符

高

低

图2.7 逻辑运算符的优先级

逻辑非“!”运算符的结合方向是“从右到左”,而“&&”和“||”结合方向则是“从左到右”。

(2) 逻辑表达式

用逻辑运算符把合法的 C 语言表达式联系起来就构成了逻辑表达式。C 语言中逻辑表达式的值也只能是 1 或 0。逻辑表达式值为 1, 则称该逻辑表达式为“真”; 若逻辑表达式值为 0, 则称该逻辑表达式为“假”。逻辑表达式的运算规则如表 2.15, 其中 a、b 为合法的 C 语言表达式。

表 2.15 逻辑表达式的运算规则

a 的值	b 的值	!a 的值	a&& b 的值	a b 的值
非 0	非 0	0	1	1
非 0	0	0	0	1
0	非 0	1	0	1
0	0	1	0	0
说 明		非 0 变 0, 0 变 1	a、b 均非 0 值才为 1	a、b 均 0 值才为 0

C 语言中, 在求解逻辑表达式、关系表达式的值时, 数字 1 代表“真”, 数字 0 代表“假”。但在判定一个表达式是否为真时, 以 0 代表“假”, 以非 0 代表“真”。例如语句:

```
a = -3.5 && 5 > 3;
```

此语句执行后, 变量 a 的值是 1。

数学上的关系式 $0 \leq x \leq 100$, 在 C 语言中不能用关系表达式 $0 \leq x \leq 100$ 来表示, 表达式 $0 \leq x \leq 100$ 相当于 $(0 \leq x) \leq 100$, 无论 x 取何值, 表达式 $0 \leq x \leq 100$ 的值总是 1, 要正确表示数学上的关系式 $0 \leq x \leq 100$, 只能用逻辑表达式 $0 \leq x \&\& x \leq 100$ 表示。

使用逻辑表达式时, 应注意逻辑表达式的“不完全计算”。例如:

```
a = 0;
b = 1;
c = a++ && b++;
d = a++ || b++;
```

对于上述第三条语句, “由左到右”扫描表达式, 根据优先级的规定, 先计算表达式 a++ 的值为 0, 变量 a 值被加 1 变成 1, 这时系统可以确定逻辑表达式 $0 \&\& b++$ 的值必定是 0, 因此不再对表达式 b++ 求值, 变量 b 的值不变仍为 1。第三句执行后变量 a、b 的值均为 1。

对于上述第四条语句, “由左到右”扫描表达式, 根据优先级的规定, 先计算表达式 a++ 的值为 1, 变量 a 的值被加 1 变成 2, 这时系统可以确定逻辑表达式 $1 || b++$ 的值必定是 1, 因此不再对表达式 b++ 求值, 因此变量 b 的值不变仍为 1。

因此执行上述程序段后, 变量 a 的值为 2, 变量 b 的值仍为 1, 变量 c 的值为 0, 变量 d 的值为 1。

3. 条件运算

C 语言中把“?:”称为条件运算符, 条件运算符要求有三个运算对象, 是 C 语言中惟一的一个三目运算符。由条件运算符构成的条件表达式的一般形式为:

判定式 ? 表达式 1 : 表达式 2

运算时先求出“判定式”的值, 若“判定式”的值是非零, 条件表达式的值取“表达式 1”的值, 若“判定式”的值为零, 条件表达式的值取“表达式 2”的值。例如以下语句执行后, 变量 min 取 a、b 中的小者,

```
min = a < b ? a : b
```

条件运算符的优先级高于赋值运算符,但低于关系运算符和逻辑运算符。条件运算符的结合方向为“从右到左”。例如:

```
a=1;b=2;
c=a<b?3:b>4?5:6;
```

上述条件表达式等价于 $a < b ? 3 : (b > 4 ? 5 : 6)$, 因此变量 c 的值为3。注意这里不等价于 $(a < b ? 3 : b > 4) ? 5 : 6$, 因为这个条件表达式的值为5, 与原意不符。

若条件表达式的<表达式1>与<表达式2>类型不同,此时条件表达式的值的类型为两者中级别较高者的类型(有关数据类型级别的高低见图2.6)。例如:

```
float f, f1;
f = (1 ? 1 : 5)/2;
f1 = (1 ? 1 : 5.0)/2;
```

上述程序段运行后,变量 f 的值为0.0,而变量 $f1$ 的值为0.5,这是因为条件表达式 $(1 ? 1 : 5)$ 的值为1,1/2为整数除,值为0,赋值给变量 f , f 值为0.0; 而 $(1 ? 1 : 5.0)$ 的值为1.0(取双精度实数类型),1.0/2为实数除,值为0.5。

2.3.4 位运算

数据在内存中都以二进制形式存放,如果对硬件编程,或作系统调用,经常需要对数据的二进制位进行操作。通常高级语言不提供位运算符,但C语言提供了位运算符,与汇编语言的位操作相似,是C语言的优点之一。

1. 位运算符

C语言提供了六种位运算符,其优先级、结合方向、要求运算对象的个数及作用如表2.16所示。

表2.16 C语言的位运算符

操作符	优先级	作用	要求运算对象的个数	结合方向
~	高 ↓ 低	按位取反	1(单目)	从右到左
<<,>>		左移,右移	2(双目)	从左到右
&		按位与	2(双目)	从左到右
^		按位异或	2(双目)	从左到右
		按位或	2(双目)	从左到右

位运算的运算对象只适用于字符型和整数型数据,对其他数据类型不适用。关系表达式和逻辑表达式的值只能是1或0,而位运算表达式的值可以是0或1以外的值。

2. 位运算符的运算规则

(1) 按位取反运算符“~”

按位取反运算符是单目运算符,运算对象在运算符的右边,其运算功能是把运算对象的内容按位取反。例如: `int i=199;`, 则 `~i` 值为-200,这是因为:

整型十进制数199的二进制数是:

```
0000 0000 1100 0111
```


把它按位取反得:1111 1111 0011 1000,这个数是整型十进制数-200在内存的补码表示。

(2)左移运算符“<<”

左移运算符的左边是运算对象,右边是整型表达式,表示左移的位数。

左移时,低位(右端)补0,高位(左端)移出部分舍弃。例如:

```
char a=5,b;
b=a<<3;
```

用二进制来表示,a 的值为0000 0101 (十进制数5),执行语句 $b=a<<3$;之后,b 的值为0010 1000 (十进制数40= $5*2*2*2$),运算后 a 的值并没有改变(仍为5)。左移时,若高位(左端)移出的部分均是二进制位数0,则每左移1位,相当于乘以2。可以利用左移这一特点,代替乘法,左移运算比乘法运算快得多。若高位移出的部分包含有二进制位数1,则不能用左移代替乘法运算。

(3)右移运算符“>>”

右移运算符的左边是运算对象,右边是整型表达式,表示右移的位数。

右移时,低位(右端)移出的二进制位数舍弃。对于正整数和无符号整数,高位(左端)补0;对于负数,高位(左端)补1(补码表示法最高位1表示负数)。例如:

```
char a=41,b;
b=a>>3;
```

用二进制来表示,a 的值为0010 1001 (十进制数41),执行语句 $b=a>>3$;之后,b 的值为0000 0101 (十进制数5= $41/2/2/2$,注意是整数除),运算后 a 的值并没有改变(仍为41)。右移时,每右移1位,相当于除以2(整数除)。可以利用左移这一特点,代替除法,右移运算比除法运算快得多。但是对于负整数,右移时高位(左端)补1,则不能用右移代替除法运算。

(4)按位与运算符“&”

运算符“&”先把两个运算对象按位对齐,再进行按位与运算,如果两个对应的位都为1,则该位的运算结果为1,否则为0。例如: $\text{int } a=41\&165$; ,则 a 的值为33,运算过程用二进制表示如下:

```
0000 0000 0010 1001    (十进制数41)
& 0000 0000 1010 0101    (十进制数165)
```

```
0000 0000 0010 0001    (十进制数33)
```

按位与运算有两个特点:和二进制位数0相与则该位被清零;和二进制位数1相与则该位保留原值不变。利用这两个特点,可以指定一个数的某一位(或某几位)置0,也可以检验一个数的某一位(或某几位)是否是1。例如 $a=a\&3$; ,只保留 a 的右端两位二进制位数。又如 $a\&4$,检验变量 a 的右端第3位是否为1。

按位与运算符“&”和逻辑与运算符“&&”不同,对于逻辑与运算符“&&”,只要两边运算数为非0,运算结果为1。例如 $41\&\&165$ 的值是1。

(5)按位异或运算符“^”

按位异或运算符“^”把两个运算对象按位对齐,如果对应位上的数相同,则该位的运算结果为0;如果对应位上的数不相同,运算结果为1。例如 $\text{int } a=41\wedge 165$; ,则 a 的值为140,运算过程用二进制表示如下:

```

0000 0000 0010 1001    (十进制数41)
^ 0000 0000 1010 0101    (十进制数165)

```

```

0000 0000 1000 1100    (十进制数140,运算时上、下相同时取0,不同时取1)

```

按位异或运算可以把一个数的二进制位的某一位(或某几位)翻转(0变1,1变0)。例如: $a = a \wedge 3$;将变量 a 的最右端的二位翻转。

(6)按位或运算符“|”

按位或运算符“|”先把两个运算对象按位对齐,再进行按位或运算,如果两个对应的位都为0,则该位的运算结果为0,否则为1。例如: $\text{int } a = 41 | 165$; ,则 a 的值为173,运算过程用二进制表示如下:

```

0000 0000 0010 1001    (十进制数41)
| 0000 0000 1010 0101    (十进制数165)

```

```

0000 0000 1010 1101    (十进制数173)

```

利用按位或运算的特点,可以指定一个数的某一位(或某几位)置1,其他位保留原值不变。例如: $a = a | 3$; ,把 a 的右端两位二进制位数置1,其他位保留原值不变。

$a = a | 0xff$; 把 a 的低字节全置1,高字节保持原样

$a = a | 0xff00$; 把 a 的高字节全置1,低字节保持原样

(7)不同数据类型之间的位运算

如果参加位运算的两个运算对象类型不同,例如长整型(long)、整型(int)或字符型(char)数据之间的位运算。此时先将两个运算对象右端对齐,若为正数或无符号数,则高位补0,负数高位补1。

例如: $9L | -200$,运算过程用二进制表示如下:

```

0000 0000 0000 0000 0000 0000 0000 1001    (十进制长整型数9L)
| 1111 1111 1111 1111 1111 1111 0011 1000    (十进制数-200,高位补1)

```

```

1111 1111 1111 1111 1111 1111 0011 1001    (十进制长整型数-199L)

```

(8)一个表达式中出现多个位运算符

如果在一个表达式中出现多个运算符时,应注意各运算符之间的优先关系。例如:语句 $a = 10 \& 5 < < 3$; 执行后 a 的值为8。“<<”的优先级高于“&”,先进行位移运算。

2.3.5 赋值运算

1. 赋值运算符和赋值表达式

C语言中“=”是赋值运算符,赋值号左边必须是变量(或是某种特定的代表存储单元的表达式,详见第五章指针),右边是一个合法的C语言表达式,由此构成赋值表达式,其一般形式如下:

变量名 = 表达式

赋值运算的功能是先计算“=”号右边表达式的值,然后把这个值赋值给“=”左边的变量,也就是把“右边表达式的值”存入“左边变量”的地址所指的存储单元中。例如表达式 $i = 123$ 是将常量123赋值给变量 i 。

赋值表达式本身有值,它的值为左边变量所得到的新值。赋值运算符的结合方向是“从右到左”。例如赋值表达式 $i=j=3*5$ 的求解过程是:先求解赋值表达式 $j=3*5$,计算右边 $3*5$ 的值为15,然后把15赋值给变量 j ,由于赋值表达式 $j=3*5$ 本身值为15,因此再求解表达式 $i=15$,把15赋值给变量 i 。这相当于用连等的方式给多个变量赋同一值。

C 语言中赋值运算符的优先级仅高于逗号运算符,低于其他运算符。

C 语言中字符类型(char)数据取值范围为 $-128\sim 127$,无符号字符类型(unsigned char)数据取值范围为 $0\sim 255$ 。例如变量 ch 与 chl 定义如下:

```
char ch;
```

```
unsigned char chl;
```

若将 ASCII 码值大于127(0x7f)的字符赋值给 ch ,它将被认为是负数。这是因为当 ASCII 码值大于0x7f 时,该字节的最高位为1,系统认为该数为负数。因此处理大于0x7f 的 ASCII 码字符(例如汉字码)时,使用 unsigned char 类型的变量 chl 较好。

2. 复合赋值运算

在赋值运算符“=”的左边加上算术运算符或位运算符就构成了复合赋值运算符。它们有十种: $*=$ 、 $/=$ 、 $\%=$ 、 $+=$ 、 $-=$ 、 $>>=$ 、 $<<=$ 、 $\&=$ 、 $\wedge=$ 、 $|=$ 。所有复合赋值运算符的运算优先级都与赋值运算符“=”的运算优先级一样,结合方向也均为“从右到左”。例如: $i-=5$ 等价于 $i=i-5$,可以理解为 $\underbrace{i}_{\uparrow} -= 5$,即把“ $i-$ ”放到等号“=”右边;

$j*=k-3$ 等价于 $j=j*(k-3)$ 可以理解为 $\underbrace{j*}_{\uparrow} = (k-3)$,这里的括号是必须的,注意不要写成 $j=j*k-3$ 。

[例2.5] 以下通过一个例子来说明复合赋值运算的用法。

```
main()
{
    int i=5;
    i+=i*i+=i+6;
    printf("i=%d\n",i);
}
```

运行结果:

$i=110$

分析表达式 $i+=i*i+=i+6$ 的执行过程:

- ①结合方向是“从右到左”,先计算 $i+6$ 的值为11;计算后 i 的值不变仍为5;
- ②再计算 $i*=11$,相当于 $i=i*11$,因此 $i=5*11$, i 的值变成55;
- ③最后计算 $i+=55$,相当于 $i=i+55$,因此 $i=55+55$, i 的值最后变成110。

3. 赋值运算中的类型转换规则

在算术赋值运算中,当赋值号右边表达式值的“数据类型”与左边“变量的类型”不一致但都是数值型时,系统将自动地把右边的数值类型转换成左边变量的类型后再进行赋值。转换规则如下:

(1)将实型数据赋值给整型变量时,舍弃实数的小数部分。例如 i 为整型变量,则执行语句 $i=56.78$;后, i 的值为56。

(2)整型数据赋值给实型变量时,数值不变,但以浮点数据形式存储到变量中。例如 f 为

float 型变量,则执行语句 $f=123;$ 时,是先将123转换成实型数据123.0000,并补足精度,再赋值给实型变量 f 。

(3)整型数据、字符型数据赋值给不同类型的整型变量、字符变量的规则如下:

①无符号 unsigned 类型与有符号类型之间的赋值转换规律

若“变量”与“数据”占内存空间的字节数相同,则进行原样赋值;若“数据”的字节数不足,则高位补0;“数据”的字节数太长,则截取低位。表2.17描述了具体转换规则。

表2.17 无符号数 unsigned 类型与有符号类型之间的赋值转换规则

本栏相同行上“变量”与“数据”占内存的字节数相同,进行原样赋值			本栏相同行上“数据”占内存的字节数太长,则截取低位;本栏相同行上“数据”占内存的字节数不足,则高位补0		
char	原样	unsigned char	char	截取低位 ← → 高位补0	unsigned int
int	←	unsigned int	char 或 int		unsigned long
long	原样	unsigned long	unsigned char		int 或 long
			unsigned int		long

[例2.6] 以下举例说明无符号 unsigned 类型与有符号类型之间的赋值转换规则。

```
main()
{
    int i=-5, j=0x9961, k;
    unsigned char ch;
    unsigned int u;
    u=i;
    ch=j;
    k=ch;
    printf("u=%x,u=%u,ch=%c,k=%d\n", u, u, ch, k);
}
```

运行结果:

$u=fffb, u=65531, ch=a, k=97$

上例的三条赋值语句的赋值过程如图2.8所示

②有符号类型与有符号类型之间的赋值转换规律

“数据”占内存的字节数不足“变量”占内存的字节数,则进行符号扩展(负数高位全补1,正数高位全补0);“数据”占内存的字节数太长,则截取低位。表2.18描述了具体转换规则。

表2.18 有符号类型与有符号类型之间的赋值转换规律

相同行上“数据”占内存的字节数太长,则截取低位; 相同行上“数据”占内存的字节数不足,则进行符号扩展		
char	截取低位 ←	int 或 long
int	→ 符号扩展	long

[例2.7] 以下举例说明有符号类型与有符号类型之间的赋值转换规则。

```
main()
```

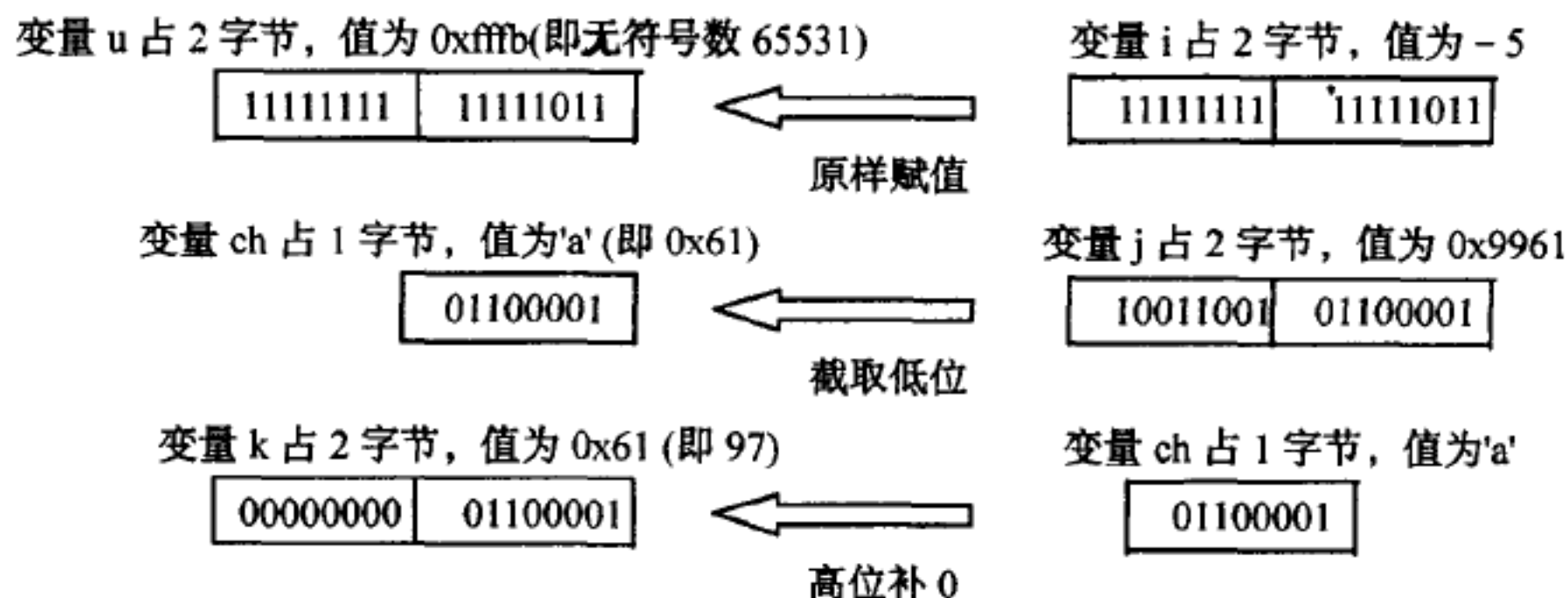



图2.8 无符号 unsigned 类型与有符号类型之间的赋值转换规则

```

{
    int i = -5, k;
    char ch;
    long int s;
    s = i;
    ch = 'a' - 32;
    k = ch;
    printf("s = %ld, ch = %c, k = %d\n", s, ch, k);
}

```

运行结果:

s = -5, ch = A, k = 65

上例中的三条赋值语句的赋值过程如图2.9所示。

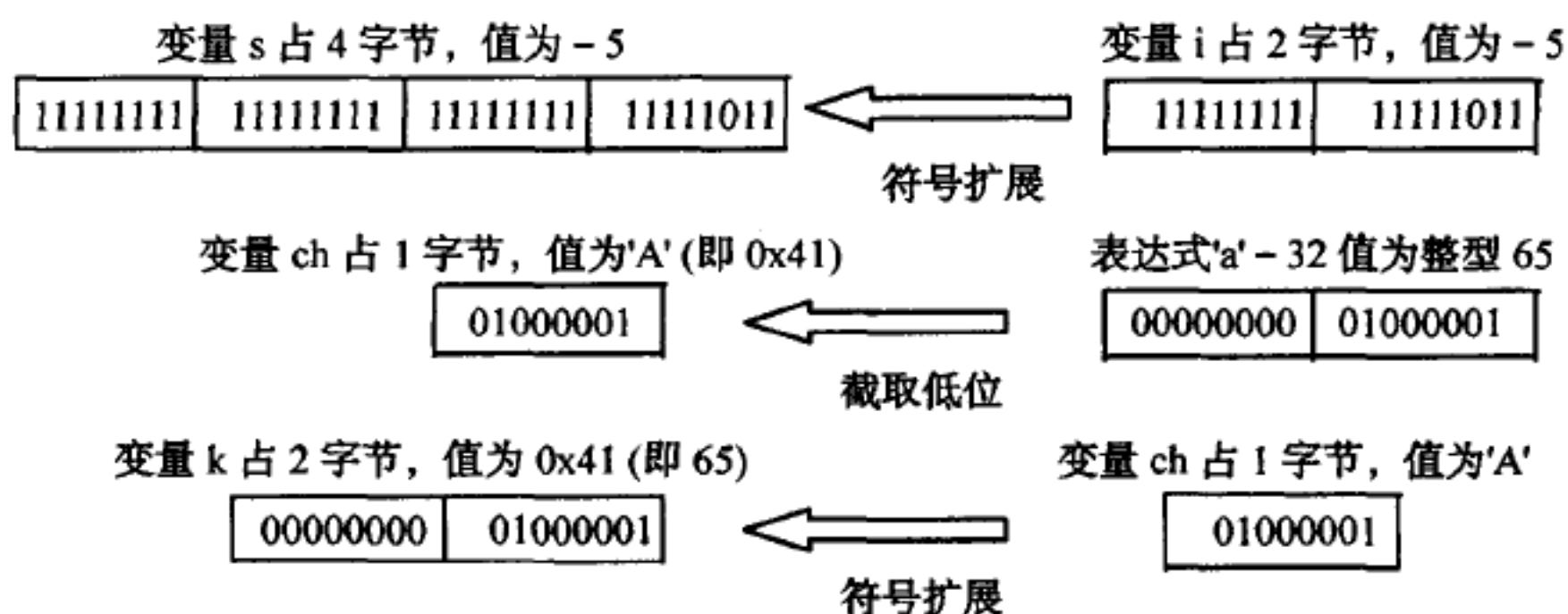


图2.9 有符号类型与有符号类型之间的赋值转换规则

2.3.6 类型转换

强制类型转换运算符将一个表达式的值转换成所需的数据类型,其使用格式为:

(类型名)(表达式)

例如(double)1/2 的值为0.5, 因为(double)1将整型数1转换成1.0(双精度类型), 1.0/2为实数除, 值为0.5。

2.3.7 逗号运算

C语言中, 可以用逗号运算符“,”把两个或多个C的合法表达式连接起来构成逗号表达式。逗号运算符的优先级是C语言中最低的。逗号表达式的一般形式为:

表达式1, 表达式2, ..., 表达式n

逗号表达式结合方向是从左至右, 先计算表达式1, 然后计算表达式2……最后计算表达式n, 逗号表达式的值为最右边表达式(即表达式n)的值。

[例2.8]

```
main()
{
    int i, j, k;
    i=1;
    k=(j=++i, i+=j, i+=5);
    printf("%d,%d,%d\n", i, j, k);
}
```

运行结果:

9,2,9

执行上面程序的第二条语句后, 变量i的初值为1。执行第三条语句时先计算表达式j=++i, i的值变为2, j的值变为2; 然后先计算表达式i+=j, 相当于i=i+j, i的值变为4; 最后计算表达式i+=5, 相当于i=i+5, i的值变为9, 并把表达式i+=5的值9作为整个逗号表达式的值赋值给变量k。最后i、j、k的值分别为9、2、9。

2.3.8 长度运算符

长度运算符“sizeof()”是单目运算符, 用于计算变量或类型所占内存字节数的大小。sizeof()运算符有两种用法:

sizeof(数据类型) 计算该数据类型在内存中所占的字节数

sizeof(变量名) 计算该变量在内存中所占的字节数

如 sizeof(int) 的值为2, sizeof(long) 值为4。可以计算变量在内存中所占的字节数, 若有语句 double d;, 则 sizeof(d) 的值为8。

2.4 小结

本章主要内容如下:

1. 标识符:关键字、预定义标识符、用户标识符;
2. 基本数据类型:整型、字符型、实型;
3. 常量:整型常量、实型常量、字符常量、字符串常量、符号常量;
4. 变量:变量的定义、变量的初始化、变量的值在内存的表示;
5. 格式输入函数 scanf、格式输出函数 printf:格式说明与使用方法;
6. 非格式化输入、输出函数:putchar、getchar、getch、getche;
7. 运算符:运算符的优先级与结合方向;
8. 表达式及求值过程:表达式的值及其类型,算术运算,增1减1运算,关系、逻辑、条件运算,位运算,赋值运算,逗号运算,长度运算,各类数值型数据的混合运算,类型转换。

习 题

一、选择题(每题只有一个正确答案)

2.1 C 语言的简单数据类型包括【1】。

- 【1】 A)整型、实型、逻辑型 B)整型、实型、字符型、逻辑型
C)整型、字符型、逻辑型 D)整型、实型、字符型

2.2 以下不属于 C 语言关键字的是【2】。

- 【2】 A)default B)unsigned C)real D)typedef

2.3 在 C 语言中,合法的字符常量是【3】。

- 【3】 A)'\\' B)"Hello!" C)'Hello' D)a

2.4 以下四组中都能正确作为 C 语言程序标识符的是【4】组。

- 【4】 A)print B)sort_3_float C)pow D)book—>name
if PI 5_abc A#B

2.5 以下 C 语言合法的数据类型关键字是【5】。

- 【5】 A)Double B)unsigned C)integer D)Char

2.6 以在 C 语言中,变量所分配的内存空间大小【6】。

- 【6】 A)均为一个字节 B)由用户自己定义 C)由变量的类型决定 D)是任意的

2.7 C 语言的字符型数据在内存中的存储形式是【7】。

- 【7】 A)原码 B)补码 C)反码 D)ASCII 码

2.8 C 语言的整型数据在内存中的存储形式是【8】。

- 【8】 A)原码 B)补码 C)反码 D)ASCII 码

2.9 C 语言中关于用户变量定义与使用的不正确描述是【9】。

- 【9】 A)变量按所定义的类型存放数据
B)系统在编译时或在运行程序时为变量分配相应的存储单元

- C)变量应先定义后使用
D)通过类型转换可更改变量存储单元的大小
- 2.10 设 int 类型的数据长度为2个字节,则 unsigned int 类型数据的取值范围是【10】。
【10】A)0至255 B)0至65535 C)-32768至32767 D)-128至127
- 2.11 C语言中整数-8在内存中存储的二进制形式是【11】。
【11】A)1111111111111000 B)1000000000001000
 C)0000000000001000 D)1111111111110111
- 2.12 下面关于C语言变量的叙述,错误的是【12】。
【12】A)变量名必须由字母或下划线开头
 B)程序中的变量必须在使用之前定义
 C)不同基本类型的变量之间可以混合运算
 D)在定义变量的同时不能对变量赋初值
- 2.13 在C语言中,合法的整型常数是【13】。
【13】A)-0x123 B)3.14159 C)01001101b D)6.7e10
- 2.14 在C语言中,合法的字符常量是【14】。
【14】A)'\101' B)"K" C)'abc' D)K
- 2.15 以下整数值中,不正确的八进制或十六进制数是【15】。
【15】A)0x16 B)-016 C)081 D)0x3A
- 2.16 下列各变量均为整型,选项中不正确的C语言赋值语句是【16】。
【16】A)i += ++i; B)i = j == k; C)i = j += i; D)i = j + 1 = k;
- 2.17 设有语句 int x=2, y=3;,则表达式 x=(y==3)的值为【17】。
【17】A)0 B)1 C)2 D)3
- 2.18 设有定义:char c; float f; int i; unsigned u; double d;,下列各表达式的类型分别为【18】。
① u+1 ② d!=f&&(i+1) ③ 4.0*i+c
【18】A)double, double, double B)int, double, char
 C)unsigned, int, double D)unsigned, unsigned, int
- 2.19 数学关系式 $x \leq y \leq z$ 可用C语言的表达式表示为【19】。
【19】A)(x<=y)&&(y<=z) B)(x<=y)and(y<=z)
 C)(x<=y<=z) D)(x<=y)&(y<z)
- 2.20 下列c为字符型变量,当且仅当c的值为小写字母时,表达式【20】为真。
【20】A)'a'<=c<='z' B)(c>=a)&&(c<=z)
 C>('a'<=c)&&('z'>=c) D)(c>='a')||(c<='z')
- 2.21 若表达式!x的值为1,则表达式【21】的值为1。
【21】A)x==0 B)x==1 C)x!=1 D)x!=0
- 2.22 下列程序段的输出结果为【22】。
int k=11, k1=-11;
printf("k=%d, k=%o, k=%x\n", k, k, k);
printf("k1=%d, k1=%o, k1=%x\n", k1, k1, k1);
【22】A)k=11, k=13, k=b B)k=11, k=11, k=b
 k1=-11, k1=177765, k1=fff5 k1=-11, k1=-13, k1=-b

C)k=11, k=11, k=11

D)k=11, k=13, k=b

k1=-11, k1=fff5, k1=b

k1=-11, k1=-13, k1=b

2.23 若 w、x、y、z 均为 int 型变量,要使以下语句的输出为1234+123+12+1,正确输入形式应当是【23】。

scanf("%4d+%3d+%2d+%1d", &x, &y, &z, &w);

printf("%4d+%3d+%2d+%1d\n", x, y, z, w);

【23】A)123412312<回车>

B)1234123412341234<回车>

C)1234+1234+1234+1234<回车>

D)1234+123+12+1<回车>

2.24 下列程序段的输出结果为【24】。

int i=-0123;

printf("i=%05d,i=%-5d,i=%u,i=%#X\n", i, i, i, i);

【24】A)i=-0083,i=-83 ,i=83,i=0XFFAD

B)i=-0083,i=-83 ,i=65453,i=0Xffad

C)i=-0083,i=-83 ,i=65453,i=0XFFAD

D)i=-0083,i=-83 ,i=-83,i=ffad

2.25 C 语言的运算符按运算对象的个数可以分为【25】。

【25】A)单目运算符一种

B)单目和双目运算符

C)单目、双目和三目运算符

D)单目、双目、三目和四目运算符

2.26 以下程序运行结果是【26】。

main()

{

int x=1, y=2, z;

z = x>y ? ++x : ++y;

printf("%d,%d,%d\n", x, y, z);

}

【26】A)1,2,3

B)1,3,3

C)2,3,3

D)2,2,3

2.27 运行下面程序段时编译时出错,其原因是【27】。

char c1='a', c2='123';

printf("%c,%d\n", c1, c2);

【27】A)字符串要用"123"表示

B)'123'只能赋值给字符数组

C)c2是字符变量,不能用%d格式输出

D)c2是字符变量,只能赋以字符常量

2.28 若定义 float a;,现要从键盘输入 a 数据,其整数位为3位,小数位为2位,则选用【28】。

【28】A)scanf("%6f", &a);

B)scanf("%5.2f", a);

C)scanf("%6.2f", &a);

D)scanf("%f", a);

2.29 若定义 double t;,则表达式 t=1, t+5, ++t 的值为【29】。

【29】A)1.0

B)2.0

C)6.0

D)7.0

2.30 已知各变量的类型说明如下,则 C 语言中错误的表达式是【30】。

int k, a, b;

unsigned long w = 5;

double x = 1.42;

【30】A) $x \% (-3)$ B) $w += 2$ C) $k = (a=2, b=3, a+b)$ D) $a += a -= (b=4 * (a=3))$

2.31 下面程序段的运行结果是【31】。

```
unsigned a = 0356, b;
b = ~a | a << 2 + 1;
printf("%x\n", b);
```

【31】A) ffba

B) ff71

C) fff8

D) fc02

2.32 下面程序段的运行结果是【32】。

```
int a=0, b=0, c=0;
if (a&&++b) c++;
printf("%d,%d\n", b, c);
```

【32】A) 1,1

B) 1,0

C) 0,0

D) 0,1

2.33 在以下运算符中,优先级最高的运算符是【33】。

【33】A) $<=$ B) $+$ C) $!=$ D) \parallel 2.34 若 $\text{int } a, b, c$; 则表达式 $(a=2, b=5, b++, a+b)$ 的值是【34】

【34】A) 7

B) 8

C) 6

D) 2

2.35 若 $\text{int } a=1, b=2, m=2, n=2$; 执行 $(m=a>b)\&\&++n$; 后, n 的值是【35】。

【35】A) 1

B) 2

C) 3

D) 4

2.36 已知各变量的类型如下,则以下符合 C 语言语法的表达式是【36】。

```
int i=8, a, b;
double x=1.42, y=5.2;
```

【36】A) $a += a -= (b=4) * (a=3)$ B) $a = a * 3 = 2$ C) $x \% (-3)$ D) $y = \text{float}(i)$ 2.37 若有以下程序段,则 z 的二进制值是【37】。

```
int x=3, y=6, z;
z = x ^ y << 2;
```

【37】A) 00011011

B) 00010100

C) 00011000

D) 00000110

2.38 在 C 语言中,判定逻辑值为“真”的最正确叙述是【38】。

【38】A) 1

B) 大于0的数

C) 非0整数

D) 非0的数

二、填空题

2.39 下面程序的功能是将 a 数据的低 4 位取反。

```
#include <stdio.h>
main()
{
    unsigned char a = 0x39, b = 【39】;
    a = a ^ b;
    printf("%d\n", a);
}
```

2.40 下面程序段运行结果是【40】。

```
int z, i=0, j=2;
```

```
z=i++&&j++;
printf( "%d,%d,%d\n", i, j, z );
```

- 2.41 运行下面程序段时,由键盘输入:12345#6.789123回车,则输出结果是【41】。

```
float f1, f2; int i, j; char ch;
scanf( " %3d%5d%c%5f%f", &i, &j, &ch, &f1, &f2);
printf( " %d, %d, %c, %f, %f\n", i, j, ch, f1, f2);
```

- 2.42 运行下面程序段,输出结果是【42】。

```
double x=4.56789;
printf( "x=%f,x=%8.3f,x=%3.8f,x=%+8.0f, x=%g,
x=%e\n", x, x, x, x, x, x);
```

- 2.43 C 语言中,标识符可分为三类,它们是【43】。

- 2.44 C 语言中,整数可用三种进制数表示,它们是【44】。

- 2.45 C 语言中,char 与 unsigned char 类型(占1个字节)的变量取值范围分别是【45】。

三、程序设计题

- 2.46 编写程序,从键盘输入三个双精度数 a、b、c,计算总和、平均值、 $x=a^2+b^2+c^2$ 的值,并计算 x 平方根的值,所有输出数据保留三位小数,第四位四舍五入。

- 2.47 输入三角形的三个边长,计算并输出三角形的面积。

- 2.48 输入两个长整型数,输出它们(整数除的)商和余数。

- 2.49 输入两个整数,输出它们(实数除的)商,并输出商的第二位小数位(例如15/8.0=1.875,1.875的第二位小数位是7)。

- 2.50 输入一个(unsigned 类型)无符号整数,分别输出它的低四位和高四位。

- 2.51 输入两个小写字母,分别赋值给字符变量 ch1 与 ch2,将它们转换成大写字母,并交换 ch1 与 ch2 的值,最后输出 ch1 与 ch2 的值。

- 2.52 输入秒数,将它按小时、分钟、秒的形式来输出。例如输入7278秒,则输出2小时1分18秒。

- 2.53 输入两个复数的实部和虚部,输出这两个复数积的实部和虚部。两复数的积按下面的公式计算: $(a+bi) \cdot (c+di) = (ac-bd) + (ad+bc)i$ 。

第 3 章

程序控制结构

C 语言是结构化程序设计语言,它强调用模块化、积木式来建立程序。采用结构化程序设计方法,可使程序的逻辑结构清晰、层次分明、可读性好、可靠性强,从而提高程序的开发效率,保证程序质量,改善程序的可靠性。

结构化程序是由三种基本结构表示的,即顺序结构、选择结构和循环结构,每种结构仅有一个入口和出口。由这三种基本结构组成的多层嵌套的程序称为结构化程序。

本章着重讨论 C 语言结构化程序的控制结构及其相应的控制语句,它们影响程序的进程。

建议本章课堂讲授 6 学时,上机 4 学时,自学 6 学时。

3.1 C 语言的语句

程序是下达给计算机的指令系列,这些指令的各种组合可以完成许多有意义的工作。在高级语言中,这些指令是通过语句来实现的。

1. 表达式语句

在 C 语言中,最常见的是表达式语句,它的格式如下:

表达式;

C 语言中允许的各种表达式后面加分号(;)组成表达式语句。分号作为语句的结束符。例如:

a=1; /* 赋值表达式 */

++i; /* 自增表达式 */

x=0,y=1; /* 逗号表达式 */

这些都是表达式加分号(;)组成的表达式语句,它们是 C 语言中最常用的语句。

函数调用也是表达式的一种形式,使用函数调用并加上一个分号(;),也就构成了一个表达式语句,如 printf("Hello! \n");等语句,我们称这种语句为函数调用语句。

在表达式语句中,有一种没有表达式,只有分号的特殊语句,称为空语句,它是 C 语言中最简单的语句。该语句不执行任何操作,常常被用来作为循环语句的空循环体(参见 3.4.1)。

2. 复合语句

复合语句是由大括号({})包围若干条语句组成的。其格式如下:

```
{  
    [数据说明部分;]  
    执行语句部分;  
}
```


在复合语句中,除了执行语句外,还可以有数据说明部分,但必须放在执行语句之前。复合语句在语法上与一个单独语句相同,凡是一个单独语句可以出现的地方都可以用复合语句。复合语句常作为循环语句的循环体和 if 语句的 if 体等。

复合语句可以嵌套,即复合语句内部还可以包含复合语句。

有关复合语句的例子将在后面的学习中结合应用加以介绍。

3. 控制语句

C 语言中有 9 种控制语句:

- | | |
|---------------|-----------------------|
| (1)if ~ else | 条件语句 |
| (2)switch | 多分支选择开关语句 |
| (3)for | 循环语句 |
| (4)while | 循环语句 |
| (5)do ~ while | 循环语句 |
| (6)continue | 结束执行循环中下面的语句,判断是否从头循环 |
| (7)break | 终止执行循环或语句 |
| (8)goto | 转向语句 |
| (9)return | 函数返回语句 |

上述 9 种语句均完成一定的程序流程控制功能。本章将介绍前面 8 种控制语句,对于 return 语句,将在第六章介绍。

3.2 顺序结构

结构化程序的最简单的结构就是顺序结构,所谓顺序结构程序就是按书写顺序执行的语句构成的程序段。

下面的简单程序仅包含了顺序结构:

[例 3.1] 计算两个键盘输入的整数的平均值并输出。

```
main()
{
    int a,b;
    float c;
    scanf("%d%d",&a,&b);
    c=a+b;
    c=c/2;
    printf("c=%f\n",c);
}
```

运行结果:

```
35    -6    /* 键盘输入 */
c=14.500000
```

上面程序的顺序结构可以用流程图(图 3.1)很方便地表示出来。

图 3.1 从(a)至(c)粗略地反映了结构化程序设计的逐步求精的设计思想,对于比较复杂

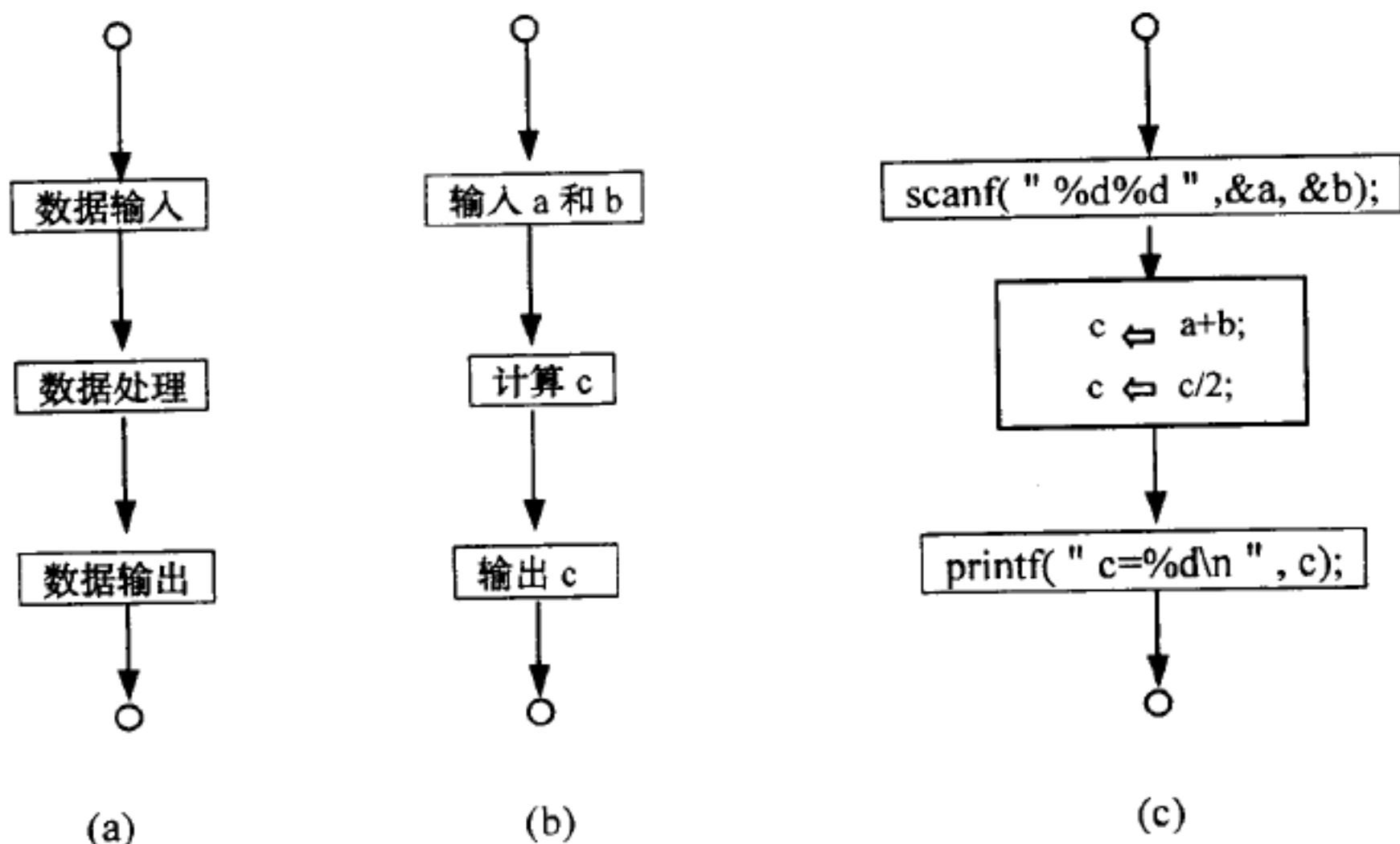


图 3.1 顺序结构的流程图

的问题,结构化程序设计一般都采用自顶向下、逐步求精的方法进行设计,这种方法思路清晰,设计严谨,也符合人们思考的习惯,而流程图可以比较直观地反映问题和设计过程。

以上流程图中,上下圆圈(○)分别表示程序(段)的入口和出口,在结构化程序设计中它们是惟一的。中间的矩形框为执行框,它包含可执行的 C 语句。线条和箭头表示程序的流程走向。

任何计算问题的答案都是按指定顺序执行一系列动作的结果,在许多场合顺序与动作同样重要,错误的执行顺序将得不到问题的正确答案。

通常情况下,程序的语句按顺序一句一句地执行,构成了顺序结构。有一些程序并不按顺序执行语句,这个过程称为“控制的转移”,它涉及了另外两类程序的控制结构,即分支结构和循环结构。

3.3 分支结构

分支结构也称为选择结构,在许多实际问题的程序设计中,根据输入数据和中间结果的不同情况需要选择不同的语句组执行,在这种情况下,必须根据某个变量或表达式的值作出判断,以决定执行哪些语句和跳过哪些语句不执行。

C 语言提供了两种类型的分支结构:

条件分支:根据给定的条件(逻辑值)进行判断,决定执行某个分支的程序段。

开关分支:根据给定整型表达式的值进行判断,然后决定执行多路分支中的一支。

条件分支主要用于两个分支的选择,由 if 语句和 if ~ else 语句来实现,开关分支用于多个分支的选择,由 switch 语句来实现。

3.3.1 if 结构

if 语句用于实现条件分支结构,它在可选动作中作出选择,执行某个分支的程序段。if 语

句有两种格式在使用中供选择。

1. if 格式(流程图见图 3.2)

如果表达式的值为真(非零值),则执行 if 语句中的语句;否则当表达式的值为假(零值)时将执行整个 if 语句下面的其他语句。if 语句中的语句可以是一条语句,也可以是复合语句。

在流程图中,我们用菱形框(也称为判断框)来表示对表达式的求值及判断,它有两路出口,分别对应菱形框中的表达式的值为真和假的两种情形。

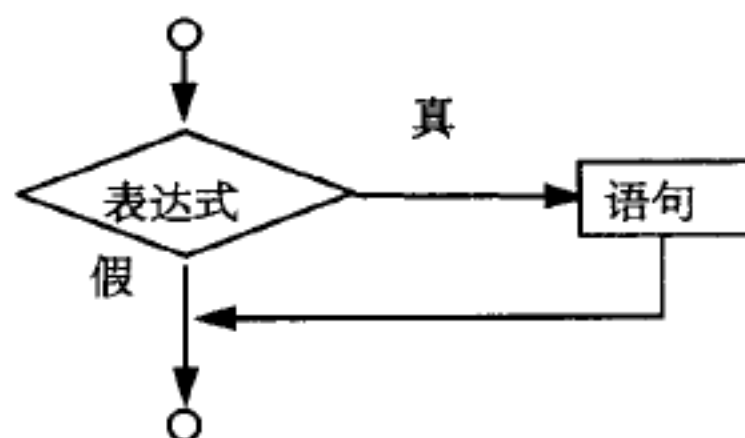


图 3.2 单分支结构流程图

[例 3.2] 求输入整数的绝对值。

```

main()
{
    int i;
    printf("请输入一个整数:");
    scanf("%d",&i);
    if(i<0)
        i=-i;
    printf("绝对值为:%d\n",i);
}
  
```

运行结果:

```

请输入一个整数:-36      /* 键盘输入 */
绝对值为:36
  
```

在上例中,当 i 为负数时才需要将它变号为正数,否则将不作任何处理,可见该语句实现的功能是在条件为真时执行某个动作,在条件为假时跳过这个动作。这种结构我们称之为“单路选择结构”。

2. if~else 格式

if~else 语句执行流程如下:

```

if(表达式)
    语句1
else
    语句2
  
```

如果表达式的值为真,则执行语句体1,否则执行语句体2。然后继续执行整个 if 语句后面的语句。语句体1和语句体2均可以是一条语句,也可以是复合语句。图3.3为 if~else 双路选择结构流程图。

[例3.3] 输入一门课程的成绩,判断是否及格。

```

main()
{
    float x;
    printf("请输入成绩:");
  
```

```
scanf("%f", &x);
if(x >= 60)
    printf("及格!\n");
else
    printf("不及格!\n");
}
```

运行结果:

请输入成绩:56 /* 键盘输入 */
不及格!

在上例中,对于输入的成绩 x 的处理有两种情况:当 x 大于等于60时将输出“及格!”,否则将输出“不及格!”。可见该语句实现的功能是在条件为真时执行某个动作,条件为假时执行另外的一个动作。这种结构也称为“双路选择结构”。

if 语句可以嵌套,即在以上两种格式的 if 语句中的语句还可以包含 if 语句。

[例3.4] 比较两个输入数的大小。

```
main()
{
    int a,b;
    printf("请输入两个整数:");
    scanf("%d%d",&a,&b);
    if(a!=b)
        if(a>b)
            printf("a>b\n");
        else
            printf("a<b\n");
    else
        printf("a=b\n");
}
```

运行结果:

99 123 /* 键盘输入 */
 $a < b$

此例中比较两个整数 a 、 b 的大小,有三种情形: $a > b$ 、 $a < b$ 和 $a = b$ 。若使用双路选择结构的 if ~ else 语句来实现就需要嵌套:① $a \neq b$ 与 $a = b$ 的双路选择;②当 $a \neq b$ 时进行 $a > b$ 和 $a < b$ 的双路选择。

注意在 if 语句中常用到比较两个数是否相等的表达式如 $a == b$,这应该是关系表达式而不是赋值表达式 $a = b$,虽然写成 $a = b$ 的程序不会产生语法错误,但显然程序在一般情况下不能产生正确的结果,这是初学者常犯的一个错误。

在嵌套 if 语句中,为了增强程序的可读性,通常把同层的 if 和 else 对齐书写,但特别需要注意的是并不是书写对齐了的 if 和 else 配对为一条 if 语句,C 语言规定 else 总是与它上面的最近的 if 配对。当 else 需要与距它较远的 if 相配对时就得使用复合语句。

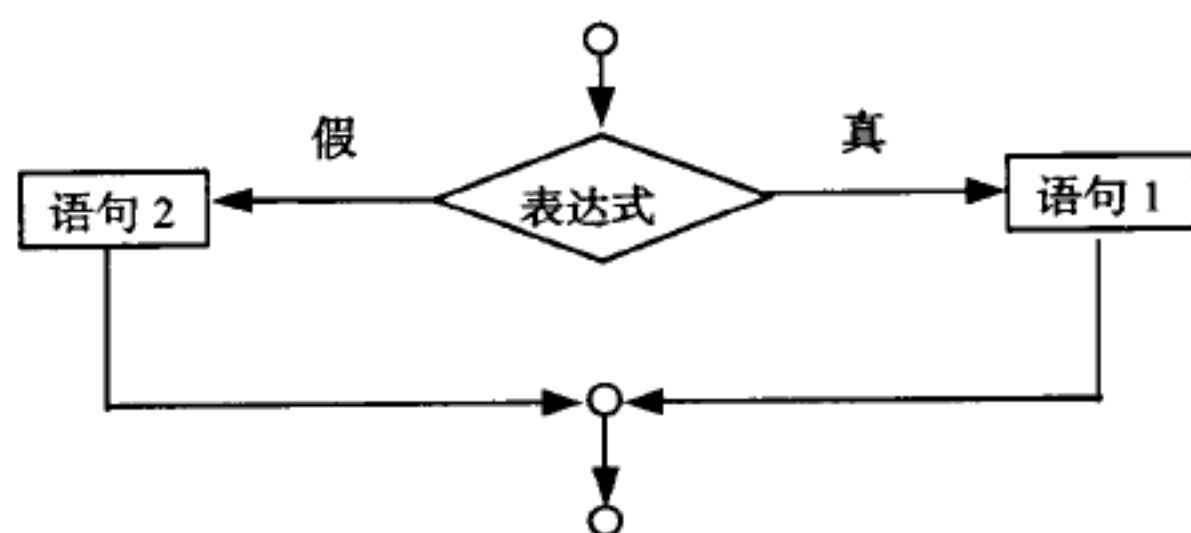


图3.3 双路选择结构流程图

例如下面两条 if 语句：

```
if(x>0)
    if(a>b)
        printf("a>b!\n");
    else
        printf("a<=b!!\n");
```

```
if(x>0)
{
    if(a>b) printf("a>b!\n");
}
else
    printf("x<=0!!\n");
```

一种 if 语句的嵌套的形式在程序设计中经常使用,它可以用来实现多路选择结构(见图 3.4),例如:

```
if(表达式1)
    语句1
else if(表达式2)
    语句2
else if(表达式3)
    语句3
    :
else if(表达式 N)
    语句 N
else
    语句 N+1
```

在图 3.4 所示程序段执行过程中,从表达式 1 开始依次计算,当遇到某个表达式 k 的值为真,则执行相应的语句 k,然后执行整个嵌套 if 语句后面的语句。如果所有的表达式的值都为假,则只执行最后一个 else 后面的语句 N+1。最后的 else 及语句 N+1 可以没有,此时若所有的表达式的值均为假的话,该 if 语句将不执行语句中的任何语句。语句 1, ..., 语句 N+1 均可以是单个语句或是复合语句。

[例 3.5] 判断从键盘输入字符的种类。将字符分为五类:

- ①控制字符(这类字符的 ASCII 码小于 32);
- ②数字字符;
- ③大写字母字符;
- ④小写字母字符;
- ⑤其他字符。

```
#include "stdio.h"
main()
{
    char c;
```

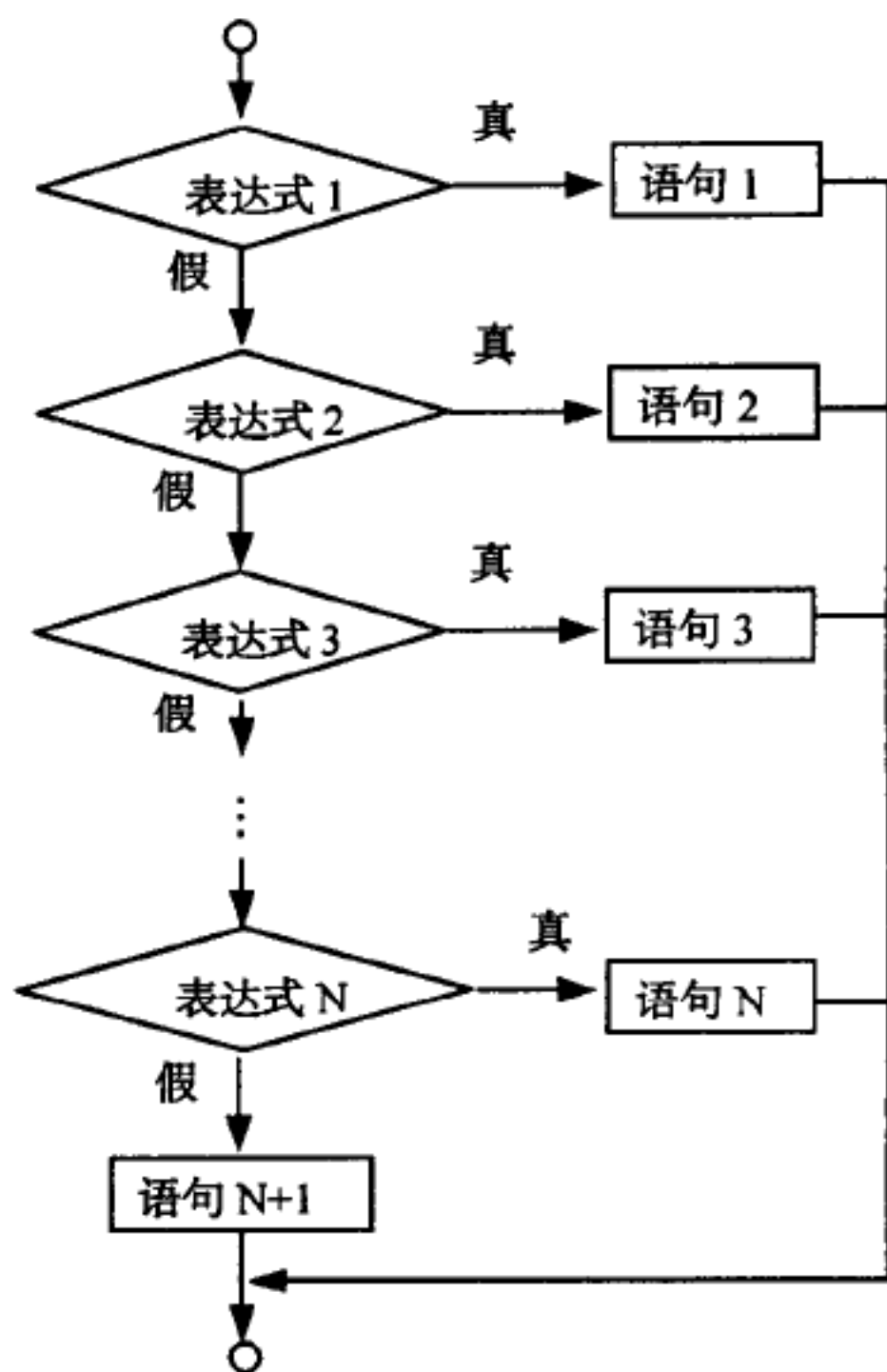


图 3.4 多路选择结构

```

printf("请输入一个字符:");
c=getche();
if (c<32)
    printf("控制字符\n");
else if (c>='0' && c<='9')
    printf("数字字符\n");
else if (c>='A' && c<='Z')
    printf("大写字母字符\n");
else if (c>='a' && c<='z')
    printf("小写字母字符\n");
else
    printf("其他字符\n");
}

```

运行结果:

```

8          /* 键盘键入 */
:数字字符
I          /* 键盘键入 */
:大写字母字符

```

上例中 if 语句含有五个语句,用于多路分支处理,对于适用于条件判断的多种平行情况的分支处理常使用 if 语句的这种形式。[例3.4]可以很方便地用这种形式改写。

对于多路分支处理还有一种情况,它对一个整型表达式的值进行计算,针对该表达式的不同取值决定执行多路分支程序段中的一支,这就是 switch 结构。

3.3.2 switch 结构

switch 结构也称为“多路选择结构”,它在许多不同的语句组之间作出选择。switch 语句用于实现该结构,它常与 break 语句联合使用,break 语句用于转换程序的流程,在 switch 语句中使用 break 语句可以使程序立即退出该结构,转而执行该结构后面的第一条语句。

switch 结构的一般格式以及结构流程图(图3.5)如下。

switch(整型表达式)

```

{
    case 整型常量表达式1:
        语句组1
        [break;]
    case 整型常量表达式2:
        语句组2
        [break;]
        :
    case 整型常量表达式 N:
        语句组 N
}

```

```

        [break;]
    default:
        语句组 N+1
}

```

在 switch 语句中,各语句组均可以由若干条语句组成。执行 switch 语句时,首先计算括号内的整型表达式的值,然后将其结果值按前后次序依次与各个 case 后面的整型常量表达式进行比较。当与整型常量表达式 k 的值一致时就执行该 case 后边的语句组 k,如果遇到 break 语句时就退出 switch 语句,继续执行该 switch 语句后面的语句;如果遇到不到 break 语句,则一直顺序往下执行下面其他 case 后边的语句,直到遇到 break 语句或最后退出 switch 语句为止。如果整型表达式的值与所有的 case 后边的整型常量表达式的值都不相等时,则执行 default 后边的语句组 N+1;若无 default 项时(default 项可缺省),则直接退出 switch 语句。

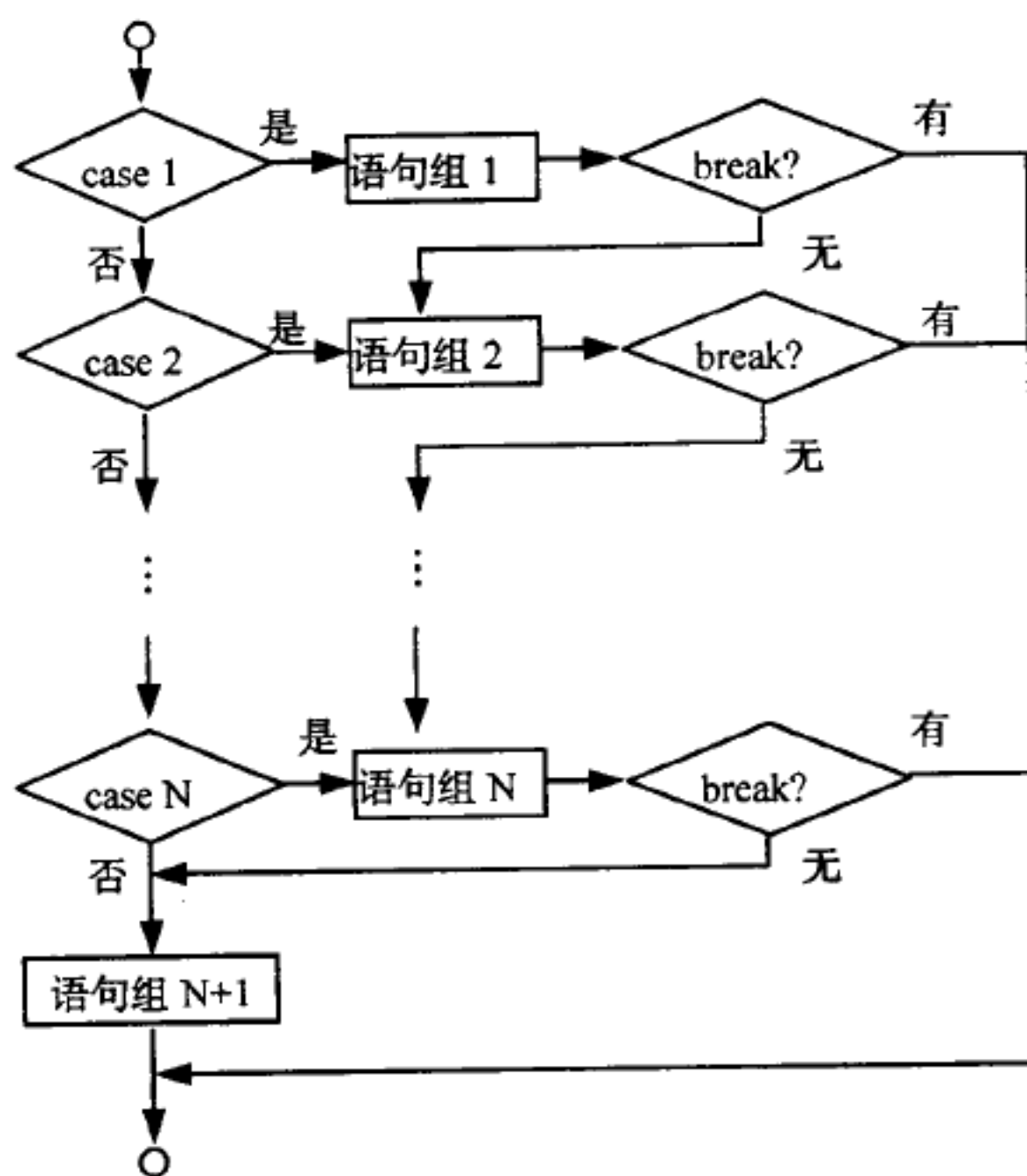


图3.5 switch 结构流程图

整型常量表达式可以是字符型常量,在计算机中字符型量是以 ASCII 码值表示的,它是一个整型值。同样在 switch 后边的整型表达式也可以是字符型变量组成的表达式,只要该表达式有一个整型值。

在 switch 语句中,语句组1, ..., 语句组 N 可以是一条语句也可以是多条语句,甚至可以没有语句。在使用 switch 语句实现多路选择结构时,通常需要适当地使用 break 语句来调整程序的流程。在某个 case 项的语句组中若需要 break 语句的话, break 语句一般都放在该语句组的最后,作为该语句组的最后一条语句,在上面 switch 语句的格式当中,为了清楚起见,我们标出了 break 语句,实际上它应该是相应语句组的一部分。

C 语言规定, case 项和 default 项在 switch 语句中能以任何顺序出现,但是把 default 项放在最后是一种良好的程序设计习惯。

在上面的流程图中,判断框“case k”(k = 1, 2, ..., N)表示整型表达式是否等于整型常量表达式的逻辑判断,它有真和假两个出口。判断框“break?”表示对在相应语句组中是否有 break 语句的两种情形的分支处理,若有该语句,则直接退出 switch 结构去执行 switch 语句后面的语句;否则程序将执行下一个语句组,进入到下一个语句组的流程。

[例3.6] 根据输入的字符 A、B、C、D、F 输出成绩“优”、“良”、“中”及“及格”和“不及格”。当输入为 F 或 R(表示缺考)时输出补考通知。(输入字符不区分大小写)

```

#include "stdio. h"

main ()
{

```

```

char grade;
printf("请输入成绩(用字母 A、B、C、D、F 表示):");
grade=getchar();
switch(grade)
{
    case 'A':case 'a':
        printf("优!\n");
        break;
    case 'B': case 'b':
        printf("良!\n");
        break;
    case 'C': case 'c':
        printf("中!\n");
        break ;
    case 'D': case 'd':
        printf("及格!\n");
        break;
    case 'F': case 'f':
        printf("不及格!\n");
    case 'R': case 'r':
        printf("请补考!\n");
        break;
    default:
        printf("成绩输入错误!\n");
}
}

```

运行结果:

```

B          /* 键盘键入 */
良!
f          /* 键盘键入 */
不及格!
请补考!

```

在上例中,switch 语句根据输入的字母是 A、B、C、D、F、R 或其他字符进行七路分支分别进行处理。由题意不区分大小写字母,程序对 case 项进行了分组,程序中所有大写字母项后面的语句组为空,根据 switch 语句的流程,它与后面的小写字母项共享一个语句组,因而具有相同的动作。注意到在 case 'F' 及 case 'f' 项的语句组中没有 break 语句,因而该语句组执行完后将继续执行其后的 case 'R' 和 case 'r' 项的语句组。这样当输入字母为 F 或 f 时,将输出两条信息:“不及格!”和“请补考!”。当输入字符在大小写英文字母 A、B、C、D、F、R 之外时,程序执行 default 项的语句组,将输出信息“成绩输入错误!”。

switch 结构用于多路分支选择,像多路开关一样,程序根据给定的整型表达式选择某一路

程序段执行,因此我们也称它为开关分支结构。因而 switch 语句也称为开关语句。

switch 语句也可以嵌套,即在语句中各 case 项的语句组可以包含另外的 switch 语句。此时 switch 结构中的 break 语句将只会使程序退出该语句所在那一层 switch 结构,并不会影响其他的 switch 结构的程序流程。

[例3.7] 嵌套 switch 语句程序示例。

```
main()
{
    int a=0,b=1;
    switch(a)
    {
        case 0: printf("a=0  ");
                switch(b)
                {
                    case 0: printf("b=0  "); break;
                    case 1: printf("b=1  "); break;
                    case 2: printf("b=2  ");
                }
        case 1: printf("a=1  ");
        default: printf("\n");
    }
}
```

运行结果:

a=0 b=1 a=1

3.4 循环结构

循环结构是程序中的另一种重要结构,它和顺序结构、选择结构共同作为各种复杂程序的基本构造部件。循环结构的特点是在给定条件成立时,反复执行某个程序段。通常我们称给定条件为循环条件,称反复执行的程序段为循环体。循环体可以是复合语句、单个语句或空语句。在循环体中也可以包含循环语句,实现循环的嵌套。

根据判定循环条件和执行循环体的先后次序,循环结构可以分为以下两种形式:

- 当型循环:首先判定循环条件,为真时将执行循环体,进行循环;否则结束循环。
- 直到型循环:首先执行循环体,再判定循环条件,为真时继续循环;否则结束循环。

3.4.1 当型循环(前判定循环)

用于实现当型循环的 C 语句有 for 语句和 while 语句,它们都能实现结构化程序设计中的循环结构,但也各有特点,使用的场合有所不同。

1. for 语句

for 语句的一般格式:

```
for(表达式1;表达式2;表达式3)
```

```
    循环体
```

for 型循环流程图如图3.6所示。

在 for 语句中,表达式1、2、3可以都有,也可以都没有,还可以有其中某一个或某两个,但 for 语句中的两个分号(;)必不可少。循环体可以是一条语句或空语句,也可以是复合语句。

for 语句的执行过程如下:

(1)首先计算表达式1;

(2)计算表达式2,以决定是否执行循环体:若表达式2的值为真(非零),则执行(3);否则跳到第(5)步,退出循环;

(3)执行循环体;

(4)计算表达式3,并转向执行(2);

(5)退出循环,继续执行循环体下面的语句。

[例3.8] 求前100个自然数之和。

```
main( )
{
    int i,sum;
    sum=0;
    for(i=1;i<=100;i++) sum+=i;
    printf("%d\n",sum);
}
```

运行结果:

5050

上例中,变量 i 在 for 循环中起到了决定什么时候退出循环、控制循环次数的作用,因此我们可以称它为循环控制变量,它出现在 for 语句的三个表达式当中。在这里,三个表达式分别起到了不同的作用:表达式1用于循环之前进行循环初始化,特别是对循环控制变量赋初值;表达式2用于表明循环的条件,即在什么条件下进行循环;表达式3用于在执行循环体一次后的某些处理,如对循环控制变量进行增量计算。这是 for 语句最典型的用法,因此我们常把 for 语句写成如下形式:

```
for(初始化;条件;增量) 循环体
```

在这种形式下,for 语句实现了自动计数器的功能。通常循环次数是确定的,它由循环控制变量的初值、增量以及循环条件给定,此时我们也称之为“定数循环”。

在 C 语言中,for 语句的功能十分强大且非常灵活,表达式1、2、3均可以省略,也可以是逗号表达式,循环体也可以是空语句。因而[例3.8]程序中的 for 语句也就可以有如下几种形式:

(1)省略表达式1:

```
i=1;
for(;i<=100;i++) sum+=i;
```

(2)省略表达式2:

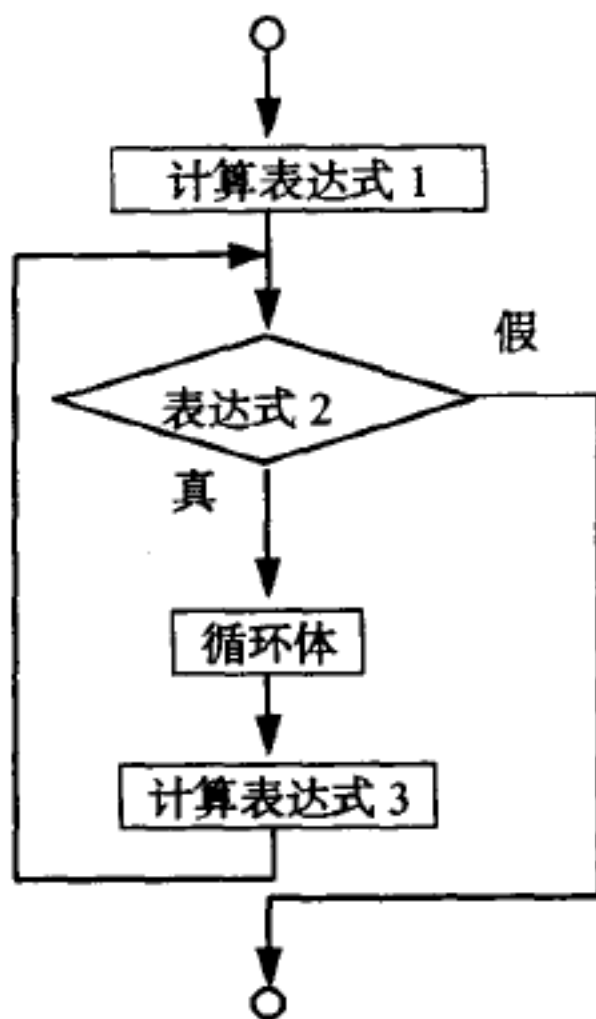


图3.6 for 型循环流程图

```
for(i=1;;i++)
{
    if(i>100) break;
    sum += i;
}
```

(3)省略表达式3:

```
for(i=1;i<=100;)
{
    sum += i;
    i++;
}
```

(4)省略循环体(即循环体为空语句),往往此时需要循环的处理在表达式3中完成:

```
for(i=1;i<=100;sum+=i,i++);
```

对于典型的定数循环建议使用“for(初始化;条件;增量)循环体”的形式,很显然语句:

```
for(i=1;i<=100;i++) sum += i;
```

要比语句:

```
for(i=1;;)
    if(i>100) break;
else
    sum += i++;
```

清晰得多,要做什么和怎么做一目了然。随着程序复杂性的增加,不规范的程序设计风格将大大降低程序的可读性,并且很容易出错,如语句:

```
for(i=1;i<=100;i++,sum+=i);
```

看上去与上面的第四种情况(省略循环体)差不多,但程序运行的结果却是不一样的。根据循环的需要,循环变量的变化可以有多种形式,例如:

(1)控制变量从100变化到1,增量为-1,循环了100次:

```
for(i=100;i>=1;i--)
```

(2)控制变量从3变化到33,增量为3,循环了11次:

```
for(i=3;i<=33;i+=3)
```

(3)控制变量从33变化到3,增量为-3,循环11次:

```
for(i=33;i>=3;i-=3)
```

(4)控制变量*i*从1开始变化,增量为*j*,循环时*j*从0开始变化,循环次数难以确定:

```
for(i=1,j=0;i<=k;i+=j,j=i-j)
```

第(4)种形式可以用来打印 Fibonacci 数列,Fibonacci 数列由下列递推公式给出:

```
fibonacci(0)=0
```

```
fibonacci(1)=1
```

```
fibonacci(n)=fibonacci(n-1)+fibonacci(n-2)
```

下面循环语句将顺序输出不大于 *k* 的 Fibonacci 数列:

```
for(i=1,j=0;i<=k;i+=j,j=i-j)
    printf(" %d ",i);
```

分析这条 for 语句不难看出在第一次循环时输出 fibonacci(1),而在以后的第 n 次循环中,i 的值为 fibonacci(n),而 j 的值为 fibonacci(n-1),n=2,3...循环过程即为 Fibonacci 数列的递推求解过程。

2. while 语句

while 循环语句的格式是:

while(表达式)

循环体

while 型循环流程图见图3.7。

该语句的执行过程是先计算表达式的值,若表达式的值为真(非零)时,执行循环体中的语句,继续循环;否则退出该循环,执行 while 语句后面的语句。循环体可以是一条语句或空语句,也可以是复合语句。

[例3.9] 统计从键盘输入的字符个数,遇到回车键时输出这个数字,结束程序。

```
#include "stdio.h"
main()
{
    int counter=0;
    while(getchar()!='\n')
        counter++;
    printf("字符数为:%d\n",counter);
}
```

程序运行:

```
5gf5er5793d456ggf123c4v56e9f /* 键盘键入 */
字符数为:28
```

在上例中,通过 getchar()函数从键盘读入一个字符,判断这个字符是否为回车符,如果不是,则进行计数工作,否则结束循环。这种循环与“定数循环”不同,程序事先并不知道循环会执行多少次,而是在程序运行过程中确定什么时候结束循环,相应地,我们可以称这种循环为“不定数循环”。这种循环通常都是通过在每次循环中获取或产生的数据与一个标记值(此例中为回车符'\n')进行比较判断,以决定是否继续循环。

循环体可以是空语句,我们来看下面的例子:

[例3.10] 求键盘输入的一个大于9的整数的最高位数字。

```
#include "stdio.h"
main()
{
    int d;
    printf("请输入一个整数(>9):");
    scanf("%d",&d);
    while((d /= 10)>9);
    printf("最高位数字为:%d\n",d);
}
```

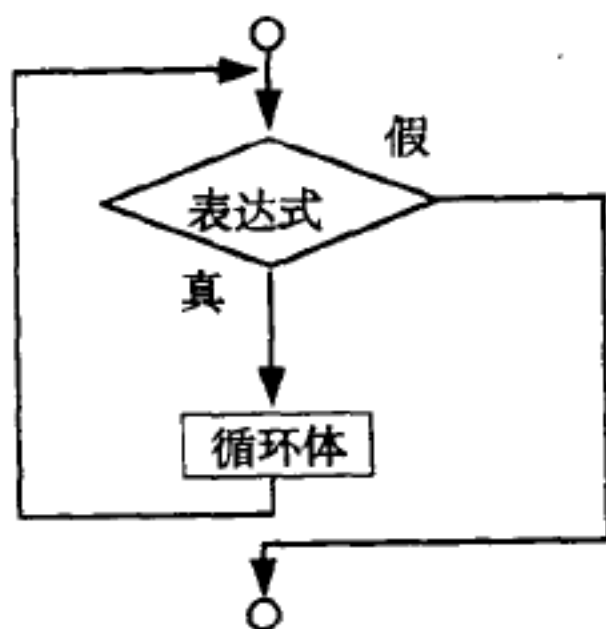


图3.7 while 型循环流程图

}

程序运行:

5678 /* 键盘键入 */

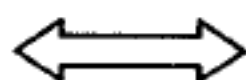
最高位数字为:5

上例中,while 语句的循环体为空语句,此时需要循环的工作放在循环条件表达式中完成。该 while 语句与下面的 while 语句是等价的:

while(d>9) d /= 10;

事实上,可以用 while 语句来实现 for 语句的循环结构:

```
for(表达式1;表达式2;表达式3)
    循环体
```



```
表达式1;
while(表达式2)
{
    循环体
    表达式3;
}
```

也可以使用 for 语句替代 while 语句:

```
while(表达式)
    循环体
```



```
for(;表达式;)
    循环体
```

只是在程序设计当中我们更多地使用 for 语句来实现定数循环而使用 while 语句来实现不定数循环,这样写的程序更清晰易懂。

3.4.2 直到型循环(后判定循环)

在当型循环中,首先判定循环的条件,若循环的条件不满足,循环体将一次也不执行。直到型循环与当型循环不同,它首先执行一次循环体,然后判定循环条件。在 C 语言中,直到型循环是由 do ~ while 语句来实现的。

do ~ while 语句的格式如下:

```
do
    循环体
while(表达式);
```

图3.8为直到型循环流程示意。

do ~ while 循环的执行过程是先执行一次循环体,然后判断表达式的值,如果是真(非零),则再执行循环体,继续循环;否则退出循环,执行下面的语句。循环体可以是单条语句或是复合语句,在语法上它也可以是空语句,但此时循环没有什么实际的意义。

为了避免与当型循环的“while(表达式);”(循环体是空语句)的形式混淆,不论循环体是否是复合语句,do ~ while 语句一般都书写成下面的形式,增加程序的清晰性和可读性。

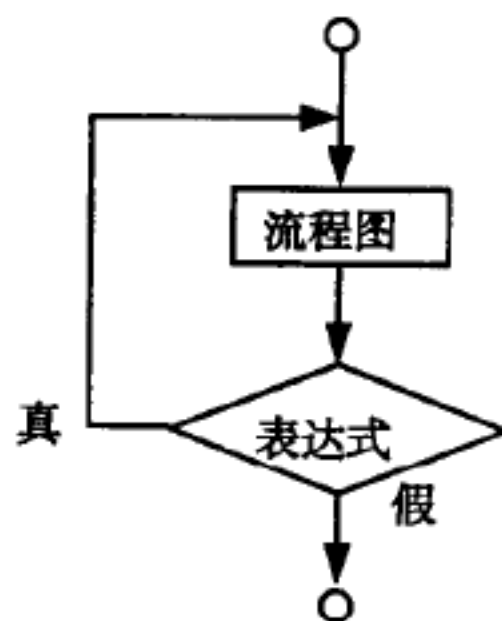


图3.8 直到型循环流程图

```
do
{
    循环体
}while(表达式);
```

[例3.11] 求整数 i , 它满足条件: $1+2+\dots+(i-1)<100$ 且 $1+2+\dots+i\geq 100$ 。

```
main()
{
    int i=0, sum=0;
    do
    {
        sum += ++i;
    }while(sum<100);
    printf("这个整数是: %d\n", i);
}
```

运行结果:

这个整数是:14

上例中, 首先执行一遍循环体得到一个和数 sum , 再判断循环条件表达式 $sum<100$, 若其值为真则继续循环, 否则退出循环。通过此例我们不难发现直到型循环通常应用于这样两种情形: ①事先我们知道初始时循环条件一定为真, 循环体肯定会被执行一遍(此例中 sum 的初值为0, $sum<100$ 一定为真); ②循环条件在循环体被执行一遍后产生, 即事先我们还不能方便地判定循环条件。请看下面例子:

[例3.12] 从键盘逐次输入一组整数对, 遇到整数对中前一个数比后一个数小时, 输出这是第几个整数对, 结束程序。

```
main()
{
    int counter=0, d1, d2;
    printf("请输入整数对:");
    do
    {
        scanf("%d%d", &d1, &d2);
        counter++;
    }while(d1>=d2);
    printf("第%d个整数对!\n", counter);
}
```

运行结果:

请输入整数对:86 78

123 96

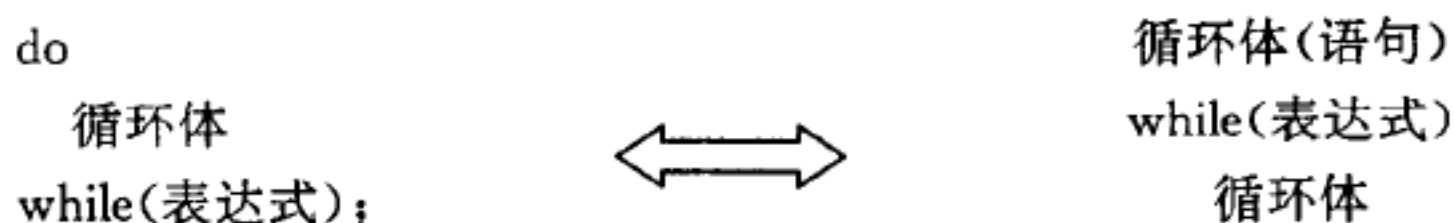
4 1

15 20

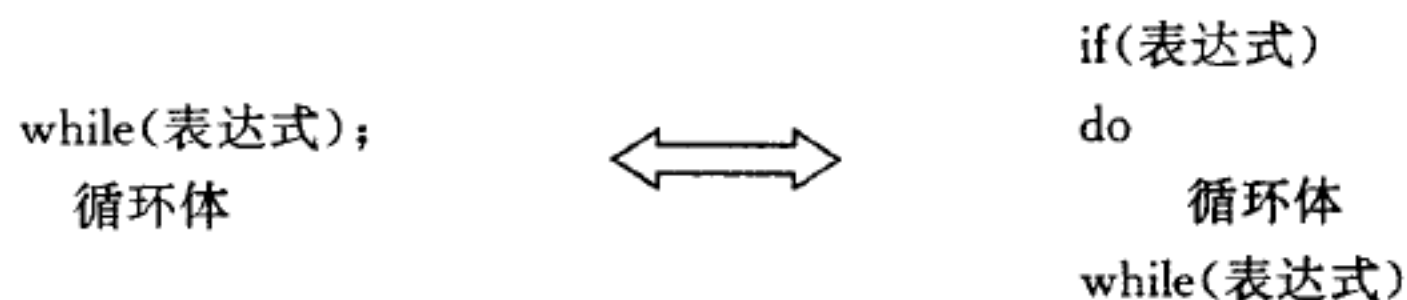
第4个整数对!

上例中,数据 d1和 d2是在循环体中输入的,因此循环条件 $d1 \geq d2$ 的判定必须在执行一次循环后进行,这时我们用直到型循环来实现它是很方便的。

直到型循环与当型循环并没有绝对的界限,事实上我们可以用当型的 while 语句来替代直到型的 do ~ while 语句:



同样我们也可以用 do ~ while 语句结合 if 语句来实现当型循环的 while 语句:



针对不同的应用场合,根据程序实现的方便程度我们可以选择不同的循环结构以及不同的循环语句,有时甚至需要多种循环语句混合及嵌套使用,形成所谓的多重循环,结合前面介绍的顺序结构和选择结构,可以写出功能非常复杂的应用程序来。下面是一个多重循环的例子:

[例3.13] 从键盘循环输入数字 n(0~9),若输入的 n 满足 $0 < n < 10$,则屏幕输出一个 $n \times n$ 的该数字构成的方阵;若输入为数字0则结束程序。

```
main()
{
    int n,i,j;
    do
    {
        printf("请输入一个数字(0~9):");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            for(j=0;j<n;j++)
                printf("%c",n+'0');
            printf("\n");
        }
    }while(n);
}
```

运行结果:

请输入一个数字(0~9):3

333

333

333

在上例中,do ~ while 语句实现程序的第一重循环,在它的循环体中完成从键盘读入数字 n,并输出 $n \times n$ 的数字 n 方阵的工作。输出方阵的工作由两条 for 语句构成的二重循环完成:内层的 for 语句实现方阵的列控制,它在方阵的每一行的 n 列个位置输出数字 n;外层的 for 语句完成方阵的行控制,它保证输出 n 行,并在内层的 for 语句打印完一行后加上一个回车符。注意当输入的 n 为 0 时,当型循环的 for 语句并不执行,因为此时外层 for 语句的表达式 $2(i < n)$ 的值为假。

另外在此例中,do ~ while 语句中的表达式为 n,一般在 if 结构及循环结构中的条件表达式多是关系与逻辑的混合表达式,其值为 1 或 0 代表真和假两个逻辑值,我们也可以用非零代表真,因此 while($n \neq 0$) 与 while(n) 是等价的,同样 while($n == 0$) 可以写成 while(!n) 这种简洁的形式。

3.4.3 break 语句与 continue 语句

break 语句和 continue 语句都是用来控制程序的流程转向的。适当地和灵活地使用它们可以更方便或更简洁地进行程序的设计。

1. break 语句

在 while、for、do ~ while 或 switch 循环体或语句组中使用 break 语句可以使程序立即退出该结构,转而执行该结构下面的第一条语句。break 语句也称为中断语句,它通常用来在适当的时候退出某个循环,或终止某个 case 并跳出 switch 结构。

[例 3.14] 从键盘输入字符、数字串,统计其中数字的个数,若遇到字符串“bye”时输出统计结果,结束程序。

```
#include "stdio.h"
main()
{
    int num=0;
    char c;
    while(1)
    {
        if((c=getche())=='b')
            if((c=getche())=='y')
                if((c=getche())=='e')
                    break;
        if(c>='0' && c<='9') num++;
    }
    printf("数字个数为:%d\n",num);
}
```

运行结果:

```
gf56694ewq2 87yr,jpaw2134598hfbye /* 键盘键入 */
```

数字个数为:15

在上例中,循环(结束)条件不能方便地用一个表达式表达出来,此时设计一个“无限循环”

结构 while(1) 形式(或 for(;;) 等形式), 在循环体中使用 if 语句构造循环结束条件, 配合 break 语句来完成这个循环结构的出口。

对于循环结构中有多个出口的情形, 也常使用 break 语句, 常见的形式如下:

```
while(1)
{
    ...
    if(表达式1) break;
    ...
    if(表达式2) break;
    ...
    ...
    if(表达式 N) break;
    ...
}
```

在这样的形式下循环可以有 N 个出口, 它们分别在循环体中的不同位置书写, 这样可以避免过大的 if 语句以及太深的 if 语句嵌套。

在一个 switch 结构中的 break 语句只会影响该 switch 结构的程序流程, 而不会影响这条 switch 语句所在的任何循环。

[例3.15] 对键盘输入的两个整数的加、减、乘、除算式进行计算, 直到用户要求退出。

```
#include "stdio.h"
main()
{
    int a,b;
    char c;
    do
    {
        printf("\n 请输入两个整数的一个算式:\n");
        scanf("%d%c%d",&a,&c,&b);
        switch(c)
        {
            case '+': printf("=%d\n",a+b); break;
            case '-': printf("=%d\n",a-b); break;
            case '*': printf("=%d\n",a*b); break;
            case '/': if(b!=0)
                        {printf("=%d\n",a/b); break; }
            default: printf("错误的算式!\n");
        }
        printf("退出程序吗?(n 或 N):");
        if(getche()=='n' || c=='N') break;
    }while(1);
}
```

```
}
```

运行结果:

请输入两个整数的一个算式:34+56

=90

退出程序吗?(n 或 N):y

请输入两个整数的一个算式:3/0

错误的算式!

退出程序吗?(n 或 N):n

上例中,switch 语句中的 break 语句将仅退出 switch 结构,循环的出口由循环体中最后一条 if 语句中的 break 语句来实现。

在进行循环结构的程序设计时,应注意循环出口的检查,保证循环能在有限步内结束,否则程序将进入“死循环”,即在运行时程序无休止地执行循环体。循环的出口可由循环条件表达式为假来完成,此时需要确认当循环了有限步后该表达式的值为假;也可以在循环体中由 if 语句及 break 语句来构造,此时需要确认循环了有限步后程序将到达 break 语句。

2. continue 语句

在 while 和 do ~ while 语句及 for 语句的循环体中,执行 continue 语句将结束本次循环而立即测试循环的条件,以决定是否进行下一次循环。

[例3.16] 计算输入的10个整数中正数的个数及平均值。

```
main()
{
    int i,n,a;
    float s;
    printf("请输入10个整数:\n");
    for(n=0,i=0;i<10;i++)
    {
        scanf("%d",&a);
        if(a<=0) continue;
        s+=a;
        n++;
    }
    printf("共有%d个正数,其平均值为%f.\n",n,s/n);
}
```

运行结果:

请输入10个整数:-1 2 3 -34 1 5 -6 4 0 -88

共有5个正数,其平均值为3.000000。

上例中对于输入的整数 a 进行分支处理:当 $a > 0$ 时进行计数和求和的工作;当 $a \leq 0$ 时则跳过这些处理,继续下一次循环。当然 continue 语句并不是必须的,将该例中 for 语句中的循环体改为:

```
scanf("%d",&a);
if(a>0)
```


```

{
    s += a;
    n++;
}

```

一样可以实现上述功能。

若在 for(表达式1; 表达式2; 表达式3) 语句的循环体中使用 continue 语句, 那么执行 continue 语句后程序将计算表达式3, 然后测试循环的条件(计算表达式2), 以决定是否进行下一轮循环, 这样就使得以下两个循环不等价:

<pre> for(表达式1; 表达式2; 表达式3) { 语句组1 continue; 语句组2 } </pre>		<pre> 表达式1; while(表达式2) { 语句组1 continue; 语句组2 表达式3; } </pre>
--	--	--

break 和 continue 语句主要用于循环体内分支比较复杂的情形, 以简化分支语句的条件, 减少条件分支语句 if 的嵌套深度及分支数, 使程序更易于阅读和理解。

[例3.17] 从键盘输入字符、数字串, 统计其中数字的个数(放在变量 num 中), 若遇到字符串“reset”时将变量 num 置零, 重新开始统计, 遇到回车符时结束程序。

```

#include "stdio.h"
main()
{
    int num=0;
    char c;
    for(;;)
    {
        if((c=getchar())=='r')
            if((c=getchar())=='e')
                if((c=getchar())=='s')
                    if((c=getchar())=='e')
                        if((c=getchar())=='t')
                        {
                            num=0;
                            continue;
                        }
        if(c=='\n') break;
        if(c>='0' && c<='9') num++;
    }
    printf("数字个数为:%d\n", num);
}

```

```
}
```

运行结果:

```
S32rr4098dfreseta94b87c372d615Z /* 键盘输入 */
```

数字个数为:10

有人认为 break 和 continue 语句破坏了结构化程序设计规范,事实上不使用 break 和 continue 语句(以及下一节介绍的 goto 语句)也能实现这些语句的效果,但若 break 和 continue 语句使用得当,并不会影响程序的可读性,并且程序的执行速度更快。

在多重循环中 break 和 continue 语句只影响该语句所在的最内层循环的程序流程,在下例中 break 语句只能跳出最内层循环。

[例3.18] 编写一个程序打印3到234之间的所有质数,每行打印10个。

程序的输出结果如下所示:

```

3      5      7      11     13     17     19     23     29     31
37     41     43     47     53     59     61     67     71     73
79     83     89     97     101    103    107    109    113    127
131    137    139    149    151    157    163    167    173    179
181    191    193    197    199    211    223    227    229    233

```

```

main()
{
    int i,j,k=0;
    for(i=3;i<234;i++)
    {
        for(j=2;j<i;j++)
            if(i%j==0)break;
        if(i==j)
            printf("%6d%c",i,++k%10==0?'\\n':' ');
    }
    printf("\\n");
}

```

此例中有两重循环,外层循环遍历了3至234之间的整数。内层循环遍历了3至*i*-1之间的整数。内层循环体测试数字*i*是否为一质数,在这里使用 break 语句可以在*i*不是一个质数时(*j*可以整除*i*)及早地中断内层循环。程序中变量*k*用于控制输出质数时每行10个。程序设计时利用更多的数学知识可以大大提高该程序的效率,如已知大于3的质数一定是奇数,因此程序中两条 for 语句可以改写为:

```
for(i=3;i<234;i+=2) 和 for(j=3;j<i;j+=2)
```

这样大约可以减少四分之三的循环次数,这在编写程序时是应该注意的。

3.5 goto 语句与标号

goto 语句配合语句标号可以实现无条件转向,进而控制程序流向。语句标号由一个有效标

标识符(标号名)加冒号(:)组成,放在某个语句之前或单独一行。同一个函数中的语句标号不能重名,同一条语句可以有几个不同的语句标号。语句标号仅对 goto 语句有意义,执行 goto 语句后程序将跳转至标号后边的语句去运行。

goto 语句的一般格式如下:

```
goto 标号名;
...
标号名: 语句
...
```

goto 语句的流程图见图3.9。

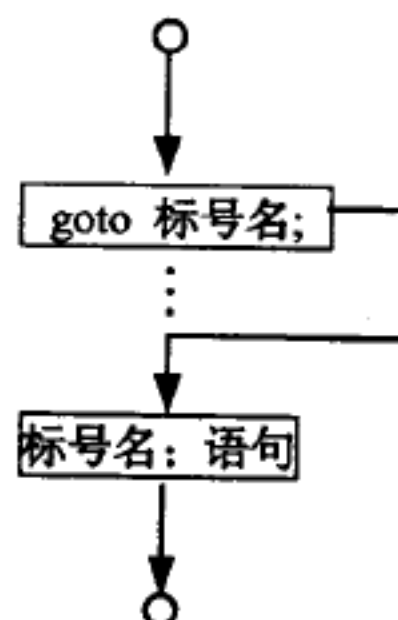


图3.9 goto 语句流程图

当程序到达“goto 标号名;”语句时,控制就转到具有该标号名的语句去执行。程序中“goto 标号名;”的位置可以在“标号名: 语句”的前面也可以在其后面,但两者必须在同一个函数体内;在一个函数体内还可以有多个 goto 语句使用同一个语句标号。goto 语句最常见的用法是用来直接退出多重循环,使用 break 语句则只能退出一层循环。另外 goto 语句也常用于程序流程转移到某个特定的地方进行特定的处理(如错误处理等)。

[例3.19] 用 goto 语句构造循环计算1到100的整数和。

```
main()
{
    int i=1,sum=0;
    loop: sum+=i++;
    if(i<=100) goto loop;
    printf("1到100的和为:%d\n",sum);
}
```

运行结果:

1到100的和为:5050

上例中,使用 goto 语句构造的直到型循环结构,运行结果同[例3.8]。

[例3.20] 这是一个智力游戏:公鸡5元一只,母鸡3元一只,小鸡1元3只,花100元买了100只鸡,问公鸡、母鸡和小鸡各有多少只?

把公鸡、母鸡和小鸡的个数记为变量 a、b 和 c,此题为求不定方程:

$$\textcircled{1} a+b+c=100$$

$$\textcircled{2} 5a+3b+c/3=100$$

的一组整数解。

```
main()
{
    int a,b,c;
    for(a=0;a<=100/5;a++) /* 遍历公鸡可能的个数 */
        for(b=0;b<=100/3;b++) /* 遍历母鸡可能的个数 */
        {
            c=100-a-b; /* a,b,c 满足不定方程① */
```

```

        if (c%3==0 && 5*a+3*b+c/3==100)
            /* 测试 a,b,c 是否满足不定方程② */
            goto end;
    }
    end: printf("公鸡=%d 母鸡=%d 小鸡=%d\n",a,b,c);
}

```

运行结果:

公鸡=0 母鸡=25 小鸡=75

上例中 goto 语句用于在找到一组解后直接跳出二重循环而到程序的出口。在程序的出口处输出这组结果。

goto 语句并不是一种必须的语言成分,不用 goto 语句一样可以实现使用 goto 语句的程序功能。不适当地使用 goto 语句往往会破坏程序的结构化风格,大大降低程序的可读性,通常情况下在程序中应尽量避免使用 goto 语句。

习 题

一、选择题

3.1 要使以下程序的输出结果为4,则 a 和 b 应满足的条件是【1】。

```

main()
{
    int s,t,a,b;
    scanf( "%d%d", &a, &b );
    s=t=1;
    if( a>0 ) s+=1;
    if( a>b ) t=s+t;
    else
        if( a==b ) t=5;
        else t=2*s;
    printf( "%d\n", t );
}

```

【1】 A) $a>0$ 并且 $a<b$

B) $a<0$ 并且 $a<b$

C) $a>0$ 并且 $a>b$

D) $a<0$ 并且 $a>b$

3.2 设有说明语句: `int a=1,b=0;`,则执行以下语句后,输出为【2】。

```

switch(a)
{
    case 1: switch(b)
        {
            case 0: printf(" ** 0 ** \n");break;
            case 1: printf(" ** 1 ** \n");break;

```

```

    }
    default: printf(" ** 2 ** \n"); break;
}

```

【2】 A) * * 0 * *

B) * * 2 * *

C) * * 0 * *

D) * * 0 * *

* * 1 * *

* * 2 * *

* * 2 * *

3.3 若 `int i;`, 则以下循环语句的循环执行次数是【3】。

```
for(i=2;i==0;) printf("%d",i--);
```

【3】 A)无限次

B) 0次

C)1次

D) 2次

3.4 在循环语句的循环体中执行 continue 语句,其作用是【4】。

【4】 A)立即终止整个循环

B)继续执行 continue 语句之后的循环体各语句

C)结束本次循环

D)结束本次循环,跳出循环

3.5 下面程序的输出结果为【5】。

```
main()
{
    int i;
    for(i=100; i<200; i++)
    {
        if (i%5==0) continue;
        printf("%d\n",i);
        break;
    }
}
```

【5】 A)100

B)101

C)无限循环

D)无输出结果

二、填空题

3.6 计算 $1+1/2+1/4+\cdots+1/50$ 的值,并显示出来。

```
main()
{
    int i=2;
    float sum=1.0;
    while( i<= 【1】 )
    {
        sum+=1/ 【2】 ;
        i+=2;
    }
    printf( "sum=%f\n", sum );
}
```

3.7 以下程序是用来统计正整数的各位数字中零的个数,并求各位数字中最大者。

```
main()
{
    unsigned long num, max, t;
    int count;
    count=max=0;
    scanf( "%ld", &num );
    do{
        t=【3】;
        if(t==0) ++count;
        else
            if( max<t )【4】;
        num/=10;
    }while( num );
    printf( "count=%d,max=%ld\n", count, max );
}
```

3.8 以下程序输出如下图形:

```

      *
     ##
    ###
   ####
  #####
 #####
#####

```

```
main()
{
    int i,j;
    for( i=1; i<=5; i++ )
    {
        for( j=1; j<=5-i; j++ ) printf( " " );
        for( j=1; j<=2*i-1; j++ )
            if(【5】) printf( " * " );
            else printf( " # " );
        printf("\n");
    }
}
```

三、改错题

3.9 以下程序段能否实现以下分段函数?若不能,请改正。

$$y = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ +1 & x > 0 \end{cases}$$

```
if (x>=0)
    if(x>0) y=1;
```



```

    y=0;
    else y=-1;

```

3.10 下面程序计算100以内正偶数之和,其中有几处错误,请改正。

```

main()
{
    int sum,i;
    for(i=100;i<=100;i--);
    if(i%2==0) sum+=i;
    else sum=0;
    printf("和为:%d\n",sum);
}

```

四、程序设计题

- 3.11 求一元二次方程 $ax^2+bx+c=0$ 的根。
- 3.12 统计键入的字符序列中的元音字母(A 或 a、E 或 e、O 或 o、I 或 i、U 或 u)的个数。
- 3.13 计算键盘输入的正整数的平均值。遇到输入数为负数时,结束程序,输出结果。
- 3.14 编程打印下列图形:

*	*****	*	*
**	*****	***	***
***	***	*****	*****
****	**	***	*****
*****	*	*	*****
(A)	(B)	(C)	(D)

3.15 编程打印十进制数1~256的二进制、八进制和十六进制数值表。

第 4 章

构造型数据类型

前面章节介绍了 C 语言中的基本数据类型(整型、字符型、实型),本章将介绍 C 语言中的构造型数据类型:数组、结构体、共用体等。讲述一维数组和多维数组的定义、初始化和使用;字符串与字符数组的概念;结构体和共用体类型变量的定义方法和使用方法,结构体和共用体的嵌套使用;枚举型的概念以及用 typedef 定义类型名。

建议本章授课 8 学时,上机 6 学时,自学 10 学时。

4.1 数组

数组是最简单、最常用的构造型数据类型,它由若干个类型相同的元素组成,每个元素就是一个变量,每个数组都有一个名字,称为数组名。数组可以是一维的,也可以是多维的。C 语言中数组元素的个数必须在定义数组时就确定,不是可调数组。

4.1.1 一维数组

1. 数组概念的引入

从以下的问题中,可以看到引入数组的必要性。键盘读入 100 个实数 a_1, a_2, \dots, a_{100} , 输出 $(a_1 + a_{100})/2.0, (a_2 + a_{99})/2.0, \dots, (a_{50} + a_{51})/2.0$ 。可以先把这 100 个实数存放在 100 实型变量中。为此定义 100 实型变量 f_0, f_1, \dots, f_{99} 分别存放这 100 个实数,程序如下:

```
scanf("%f", &f0);
scanf("%f", &f1);
    ⋮
scanf("%f", &f99);
printf("%f, ", (f0+f99)/2.0);
printf("%f, ", (f1+f98)/2.0);
    ⋮
printf("%f, ", (f49+f50)/2.0);
```

上述程序重复、繁琐。为了简化这一程序,引入数组概念,将这 100 个实型变量写成 $f[0], f[1], \dots, f[99]$, 它们都有相同的数组名 f , 其中方括号中的数字称为数组下标, 数组下标可以用整型变量或整型表达式来表示。设 i 为整型变量, 当 $i=0$ 时 $f[i]$ 就代表 $f[0]$, \dots , 当 $i=99$ 时 $f[i]$ 就代表 $f[99]$ 。因此, 上述程序简化为:

```
for( i=0; i<100; i++) scanf("%f", &f[i]);
```

```
for( i=0; i<50; i++) printf(" %f, ", (f[i]+f[99-i])/2.0);
```

可见,引入数组之后,程序变得非常简练。本章讲述的排序和查找等程序设计算法,不使用数组就无法进行。程序设计语言中,数组是必不可少的重要数据类型。

2. 一维数组的定义

一维数组是指数组元素只有一个下标的数组。定义一维数组的一般格式如下:

类型名 数组名[整型常量表达式],...;

类型名即数据类型标识符,是每一个数组元素的数据类型,可以是整型、实型、字符型、指针类型、结构体和共用体等。整型常量表达式的值就是该数组元素的个数。例如下述定义:

```
int a[10];
```

这条定义语句说明了:

(1)定义了一个名为 *a* 的一维数组。

(2)方括号中的 10 规定了 *a* 数组有 10 个元素,它们是 *a*[0],*a*[1],*a*[2],...,*a*[9]。

(3)类型名 *int* 规定了 *a* 数组的每个元素都是整型变量。

(4)每个元素只有一个下标,因此数组 *a* 是一维数组。C 语言规定数组的第一个元素的数组下标(数组下标的下界)是 0。在 C 语言中一个数组最后一个元素的下标(数组下标的上界)是数组元素的个数减 1。例如上述 *a* 数组下标的上界为 9,则最后一个元素为 *a*[9]。

(5)系统将为 *a* 数组在内存中分配一块 20 个字节的连续存储单元,每个元素(*int* 类型)占 2 个字节,*a*[0]元素分配的单元地址最低,*a*[9]元素分配的单元地址最高,如图 4.1 所示。

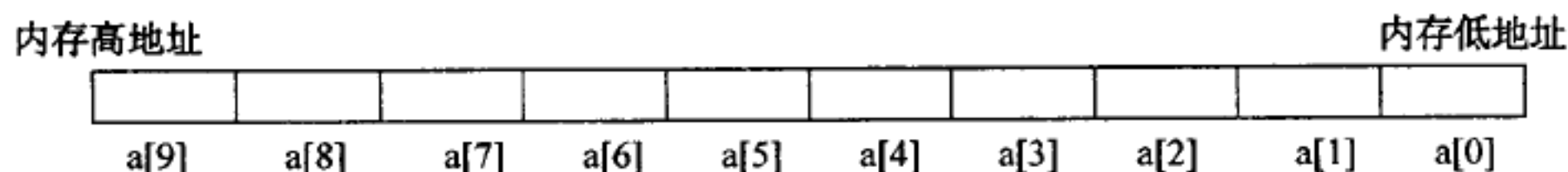


图 4.1 一维数组 *a* 的元素分配存储单元情况

一条语句中,可以定义多个数组,也可以在定义普通变量名的同时定义数组。例如:

```
float x1, y[100], x2, z[10];
```

以上语句定义了两个单精度实型变量 *x1* 与 *x2*,同时还定义了两个单精度实型数组 *y* 与 *z*,其中 *y* 数组包含 100 个元素,它们是 *y*[0]~*y*[99];*z* 数组包含 10 个元素,它们是 *z*[0]~*z*[9]。

3. 一维数组元素的使用

数组定义后,就可以在程序中使用数组元素,一维数组元素的使用格式如下

数组名[下标表达式]

此处下标表达式可以是整型常量表达式或整型表达式。例如有如下定义语句:

```
float f[10];
```

设变量 *i*、*j* 均为整型变量,则对数组 *f* 的元素合法的使用形式可以是 *f*[0],*f*[1],...,*f*[9],*f*[*i*],*f*[*i*+*j*],*f*[*i*-5]等,程序中可以像使用变量一样使用这些元素。例如下述语句都是正确的:

```
scanf(" %f%f", &f[3], &f[5]);
```

```
f[1]=(f[3]+f[5])/5.0;
```

```
printf(" %f\n", f[1]);
```

由于定义数组 *f* 有 10 个元素,因此下标表达式的值必须大于等于 0,并且小于等于 9。

使用数组元素应该注意:

(1)系统在内存中为数组分配一块连续的存储单元,最低的地址对应于第一个数组元素,最高的地址对应最后一个数组元素。每个数组元素都等同于一个变量,使用数组元素就可以存取这些存储单元内的数据,就像存取变量的值一样。

(2)C语言中,不能对一个数组整体赋值。例如若有下述程序段:

```
int a[10], b[10], i;
for( i=0; i<10; i++) scanf("%d", &a[i]);
```

则语句 `b=a;` 是非法的,要把 `a` 数组元素赋值给下标对应的 `b` 数组元素可用如下语句:

```
for(i=0;i<10;i++)b[i]=a[i];
```

(3)在使用数组元素时,数组元素中下标表达式的值必须是整型,下标表达式值的下限为0,下标表达式值的上限为该数组元素的个数减1。C语言程序在运行过程中,系统并不自动检验数组元素的下标是否越界。因此数组两端都可能因为越界而破坏了其他存储单元中的数据,甚至破坏程序代码或操作系统。因此,在编写C语言程序时,应该特别注意保证数组下标不越界。

4. 数组元素的初始化

(1)定义数组时不对元素赋初值

①动态数组

上述定义的数组均为动态数组,定义动态数组时不对元素赋初值,数组元素的值没有确定的值。例如定义:

```
int a[10];
```

则 `a[i]` (其中 `i=0,1,...,9`) 的值是不确定的值,可以是 `-32768~32767` 之间的任意一个整数。

②静态数组

定义数组时不对元素赋初值,所有元素初值为0。例如定义:

```
static int b[10];
```

则 `b[i]` (其中 `i=0,1,...,9`) 的值是0。这里 `int` 前面有一个关键字 `static`,它是“静态存储”的意思(有关静态存储的概念将在第六章中详细介绍),静态存储变量的初始化是在编译阶段完成的,而动态变量的初始化是在运行程序时才完成。

(2)定义数组时对元素赋初值

Turbo C 中不仅可以对静态数组进行初始化,也允许对动态数组进行初始化,可以采用以下形式,在定义语句中为所定义的数组元素赋初值:

```
[static] 类型名 数组名[整型常量表达式]={常量1,常量2,...,常量N};
```

省略关键字 `static`,定义的数组就是一个动态数组,例如下面对一个动态数初始化:

```
int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

所赋初值放在赋值号后的一对花括号中,初值的类型必须与所说明的类型一致,所赋初值之间用逗号隔开,系统将按这些数值的排列顺序,从 `a[0]` 元素开始依次给 `a` 数组中的元素赋初值,即 `a[0]` 赋初值1, `a[1]` 赋初值2, ..., `a[9]` 赋初值10。在指定初值时,第一个初值必定赋给下标为0的元素,因此,不可能跳过前面的元素给后面的元素赋初值。

应该注意:

①如果初值的数据个数比数组元素少,则自动给后面的元素补赋初值0。例如,以下定义语句将给 `a` 数组中所有元素赋初值0。


```
int a[10]={0};
```

再如:

```
int b[5]={1,2};
```

则 $b[0]$ 赋初值 1, $b[1]$ 赋初值 2, $b[2]$ 、 $b[3]$ 、 $b[4]$ 均赋初值 0。

②如果所赋初值多于所定义数组的元素个数时,在编译时将提示出错信息:

Too many initializers。

5. 初始化时不指定数组长度

C 语言中可以在初始化时,不指定数组元素的个数。此时数组元素的个数等于所赋初值数据的个数。例如:

```
int a[]={1, 2, 3, 4, 5};
```

这里花括号中列出 5 个初始化数据,它隐含地定义了 a 数组共有 5 个元素,等价于语句:

```
int a[5]={1, 2, 3, 4, 5};
```

6. 一维数组程序举例

排序问题是程序设计中的典型问题之一,所谓排序就是将数组中的各元素的值按从小到大(或从大到小)的顺序重新排列。排序过程一般都要进行元素值的比较和元素值的交换。下面举例说明两种排序算法:冒泡排序法和选择排序法。

[例 4.1] 用冒泡排序法对键盘输入的 8 个整数从小到大进行排序。

冒泡排序法的基本思想:假设有 n 个数放在数组 a 中,现要把这 n 个数从小到大排序。首先,在 $a[0]$ 到 $a[n-1]$ 的范围内,依次比较两个相邻元素的值,若 $a[j]>a[j+1]$,则交换 $a[j]$ 与 $a[j+1]$,否则不交换, $j=0,1,2,\dots,n-2$,经过这样 $n-1$ 次比较(称为一趟冒泡),就把 $a[0]\sim a[n-1]$ 中最大的值换到了元素 $a[n-1]$ 中;然后在 $a[0]\sim a[n-2]$ 的范围内再进行一趟冒泡,又将该范围内元素的最大值换到了元素 $a[n-2]$ 中;依次进行下去,最多只要进行 $n-1$ 趟冒泡,就可完成排序。如果在某趟冒泡过程中没有交换相邻元素的值,则说明排序已完成,可以提前结束处理。如图 4.2 所示的是一个冒泡排序的例子,图中用花括号表示每趟冒泡的范围。

{6,8,5,4,6,9,3,1} 第 1 趟冒泡:比较 7 次相邻元素的值,若 $a[j]>a[j+1]$,则交换 $a[j]$ 与 $a[j+1]$,最后把花括号范围内最大的值 9 换到 $a[7]$,数组元素排列如下:

{6,5,4,6,8,3,1},9 第 2 趟冒泡:比较 6 次相邻元素的值,若 $a[j]>a[j+1]$,则交换 $a[j]$ 与 $a[j+1]$,最后把花括号范围内最大的值 8 换到 $a[6]$,数组元素排列如下:

{5,4,6,6,3,1},8,9 第 3 趟冒泡后数组元素排列如下:

{4,5,6,3,1},6,8,9 第 4 趟冒泡后数组元素排列如下:

{4,5,3,1},6,6,8,9 第 5 趟冒泡后数组元素排列如下:

{4,3,1},5,6,6,8,9 第 6 趟冒泡后数组元素排列如下:

{3,1},4,5,6,6,8,9 第 7 趟冒泡后数组元素排列如下:

1,3,4,5,6,6,8,9 排序完成。

图 4.2 对 a 数组中的 $a[0]$ 到 $a[7]$ 8 个元素用冒泡算法排序

根据上述算法,写出程序如下:

```
main()
{
    int i, j, a[8], temp, swap;
```

```

/* 变量 swap 用于判定本趟冒泡是否交换了元素值 */
printf("请输入 8 个整数:");
for(i=0; i<8; i++)
    scanf("%d", &a[i]);
for(i=0; i<8-1; i++)
{
    swap=0;
    for(j=0; j<8-i-1; j++)
        if( a[j]>a[j+1] )
        {
            swap=1;
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }
    if( !swap ) break;
}
printf("8个数排序后的结果为:");
for(i=0; i<8; i++)
    printf("%3d", a[i]);
printf("\n");
}

```

运行结果:

请输入8个整数:6 8 5 4 6 9 3 1

8个数排序后的结果为: 1, 3, 4, 5, 6, 6, 8, 9,

[例4.2] 用选择排序法对键盘输入的8个整数从小到大进行排序。

选择排序法的基本思想:假设有 n 个数放在数组 a 中,现要把这 n 个数从小到大排序。首先,在 $a[0]$ 到 $a[n-1]$ 的范围内,选出值最小元素与 $a[0]$ 交换;然后在 $a[1] \sim a[n-1]$ 的范围内再选出值最小的元素与 $a[1]$ 交换;依次进行下去,进行 $n-1$ 次选择后就可完成排序。

从 $a[i] \sim a[n-1]$ 中选出值最小的元素并与 $a[i]$ 交换位置,可用下列步骤实现:引入整型变量 j 和 k ,变量 j 赋初值 $i+1$,变量 k 赋初值 i ;反复比较 $a[k]$ 和 $a[j]$ 的值,若 $a[k] > a[j]$ 则将 j 值赋给 k (即 k 总是记录着值最小的元素下标),每进行一次比较, j 加1;当 $j > n-1$ 时,若 $k \neq i$,则交换 $a[i]$ 与 $a[k]$ 。

显然选择排序法对冒泡排序法进行了改进,它减少了元素之间交换数据的次数。

图4.3是一个选择排序的例子,图中用花括号表示每次选择的范围。

{6,8,5,4,6,9,3,1} 第1次选择:选出花括号范围内最小值1与 a[0]交换;
 1,{8,5,4,6,9,3,6} 第2次选择:选出花括号范围内最小值3与 a[1]交换;
 1,3,{5,4,6,9,8,6} 第3次选择:选出花括号范围内最小值4与 a[2]交换;
 1,3,4,{5,6,9,8,6} 第4次选择:选出花括号范围内最小值5与 a[3]交换;
 1,3,4,5,{6,9,8,6} 第5次选择:选出花括号范围内最小值6与 a[4]交换;
 1,3,4,5,6,{9,8,6} 第6次选择:选出花括号范围内最小值6与 a[5]交换;
 1,3,4,5,6,6,{8,9} 第7次选择:选出花括号范围内最小值8与 a[6]交换;
 1,3,4,5,6,6,8,9 排序完成。

图4.3 对 a 数组中的 a[0]到 a[7]8个元素用选择算法排序

根据上述算法,写出程序如下:

```
main()
{
    int i, j, k, a[8], temp;
    printf("请输入8个整数:");
    for(i=0; i<8; i++)
        scanf("%d", &a[i]);
    for(i=0; i<8-1; i++)
    {
        k = i;
        for(j=i+1; j<8; j++)
            if( a[k]>a[j] ) k=j;
        if( k != i )
        {
            temp=a[i];
            a[i]=a[k];
            a[k]=temp;
        }
    }
    printf("8个数排序后的结果为:");
    for(i=0; i<8; i++)
        printf("%3d,", a[i]);
    printf("\n");
}
```

运行结果:

请输入8个整数:6 8 5 4 6 9 3 1

8个数排序后的结果为: 1, 3, 4, 5, 6, 6, 8, 9,

[例4.3] 有8个数按从小到大的顺序存放在一个数组中,输入一个数,用折半法找出该数是数组中第几个元素的值。如果该数不在数组中,则输出“没找到”。

用折半法在某数组中查找一个数,要求该数组已经作了排序。假设有 n 个元素按从小到大的顺序存放在数组 a[0]~a[n-1]中,需要查找的数是 x,引入两个整型变量 low 与 high 分别

表示查找区间两端点元素的下标。首先 $low=0$, $high=n-1$, 即开始是在 $a[0] \sim a[n-1]$ 区间内查找。设 mid 为一整型变量, 查找时, 令 $mid=(high+low)/2$, 比较 x 与 $a[mid]$ 的值, 有三种情况:

①若 x 等于 $a[mid]$, 则说明找到了。

②若 $x > a[mid]$, 说明待查元素可能在 $a[mid+1] \sim a[high]$ 之间, 可令 $low=mid+1$ 。

③若 $x < a[mid]$, 说明待查元素可能在 $a[low] \sim a[mid-1]$ 之间, 可令 $high=mid-1$ 。若是②、③两种情况, 则查找范围缩小了一半。重复上述查找过程, 直到查找范围缩小到零(没找到), 或 x 等于 $a[mid]$ (查找成功) 为止。

图4.4是两个折半查找的例子, 图中用花括号表示每次查找的区间, 箭头指向中间元素 $a[mid]$ 。

{6, 9, 15, 25, 26, 36, 48, 53} 第1次查找: $low=0$, $high=7$, $mid=(0+7)/2=3$;

↑

6, 9, 15, 25, {26, 36, 48, 53} 第2次查找: $low=4$, $high=7$, $mid=(4+7)/2=5$;

↑

6, 9, 15, 25, {26}, 36, 48, 53 第3次查找: $low=4$, $high=4$, $mid=(4+4)/2=4$ 。

↑

(a) 折半查找值为26的元素, 经过三次比较查找成功

{6, 9, 15, 25, 26, 36, 48, 53} 第1次查找: $low=0$, $high=7$, $mid=(0+7)/2=3$;

↑

{6, 9, 15}, 25, 26, 36, 48, 53 第2次查找: $low=0$, $high=2$, $mid=(0+2)/2=1$;

↑

6, 9, {15}, 25, 26, 36, 48, 53 第3次查找: $low=2$, $high=2$, $mid=(2+2)/2=2$;

↑

6, 9, {}, 15, 25, 26, 36, 48, 53 此时 $low=3$, $high=2$, 即查找范围缩小到零, 没找到。

(b) 折半查找值为11的元素, 经过三次比较查找范围缩小到零, 没找到

图4.4 折半查找

根据以上分析, 编写程序如下:

```
#define N 8
main()
{
    int a[N]={ 6, 9, 15, 25, 26, 36, 48, 53 };
    int low=0, high=N-1, mid, x, found=0;
    /* 变量 found 用于判定是否找到 */
    printf("请输入查找数据:");
    scanf("%d", &x);
    if( x >= a[low] && x <= a[high]) /* 如果 x 位于查找区间内, 则开始查找 */
        while(!found && low <= high)
        {
            mid = (low + high) / 2;
```



```

        if (a[mid] == x) found=1;
        else if(x > a[mid]) low=mid+1;
        else high=mid-1;
    }
    if( found ) printf("元素值为 %d 的数组下标是 %d.\n", x, mid);
    else printf("未找到.\n");
}

```

运行结果:

请输入查找数据:26

元素值为 26 的数组下标是 4。

4.1.2 字符数组

C 语言中没有字符串类型的变量,字符串的存储有两种方式,一是把字符串存储在一个字符数组中,第二种方法是利用指针处理字符串常量(详见第五章)。本节介绍使用字符数组来处理字符串的方法。

1. 字符数组的定义

字符数组的每个元素存放一个字符,其定义方法与定义一个一维数组相同:

```
char 数组名[整型常量表达式],...;
```

例如:char str[15];

此语句定义了一个有15个元素的字符数组 str。

2. 字符数组的初始化

(1) 字符数组逐个元素赋初值

这种情况与一般的数组赋初值方法相同。例如:

```
char s[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
```

上述语句等同于:char s[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};

(2) 在赋初值时直接赋字符串常量

可以直接用字符串常量给一维字符数组赋初值。例如:

```
char s[] = { "Hello!" };
```

赋初值时,若不给定数组元素的个数,系统将按字符串中实际的字符个数来定义数组的大小。上述语句等同于 char s[7] = { "Hello!" };。用字符串常量给字符数组赋初值时,系统自动在最后补上字符串结束标志'\0',所以不必人为加入。赋值时也可以省略花括号,简写成:

```
char s[] = "Hello!";
```

这里字符数组 s 的内容如图4.5所示。

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]
H	e	l	l	o	!	\0

图4.5 用字符数组存放字符串

用字符串常量直接给字符数组赋初值,应注意字符数组要有足够的存储单元存储字符串。例如:char str[6] = "Hello!";该语句定义了一个有6个元素的字符数组 str,但是字符串常

量"Hello!"包括了字符串结束标志'\0',要占用7个存储单元,6个单元空间不够用,'\0'将占用下一个不属于字符数组 str 的存储单元,会破坏其他内存单元的数据或程序代码。

3. 给字符数组元素逐个赋字符值

定义了一个字符数组后,就可以对其元素赋值。例如:

```
char s[9];
```

```
s[0]='H'; s[1]='e'; s[2]='l'; s[3]='l'; s[4]='o'; s[5]='!';
```

上述语句把6个字符存入了字符数组 s,如图4.6所示。

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]
H	e	l	l	o	!			

图4.6 用字符数组存放字符

以上字符数组 s 中存放了6个字符,并不等同于 s 中存放了字符串"Hello!"。因为 C 语言规定字符串要以字符串结束标志'\0'结束。上面若加上 s[6]='\0';,才把字符串"Hello!"存入字符数组 s 中。因此,用字符数组来存放字符串时,若是逐个字符赋值给数组元素,要在最后一个字符之后加上字符串结束标志'\0'。

用逐个字符给字符数组赋值,对字符串的存取不太方便。C 语言提供了专门用于处理字符串的各种库函数。利用字符串处理函数,可以方便地将字符串存入字符数组,方便地输入、输出字符串。

4. 利用库函数 strcpy 给字符数组赋字符串

(1) 不能用赋值语句把字符串整体赋值给字符数组

在 C 语言中,不能用赋值语句把字符串整体赋值给字符数组。例如:

```
char s1[30], s2[30], s3[ ]="Good Morning!";
```

```
s1="Hello!";
```

```
s2=s3;
```

以上两句赋值语句都是非法的。第一个赋值语句相当于把字符串"Hello!"的首地址赋值给 s1,第二个赋值语句相当于把数组 s3 的首地址赋值给 s2,但 s1 和 s2 都是数组的首地址,是地址常量,不能被赋值。

(2) 利用库函数 strcpy 给字符数组赋字符串

可以方便地利用库函数 strcpy 把字符串存入字符数组,函数 strcpy 的调用形式为:

```
strcpy(字符数组1,字符串2);
```

strcpy 的作用是将字符串2复制到字符数组1中,复制时将字符串2后面的字符串结束标志'\0'也复制到字符数组1中。例如下面的程序段:

```
char str1[30], str2[30], str3[ ]="How are you.";
```

```
strcpy(str1, str3); /* 将字符串 "How are you." 存入字符数组 str1 中 */
```

```
strcpy(str2, "How do you do."); /* 将字符串 "How do you do." 存入字符数组 str2 中 */
```

使用函数 strcpy 应该注意字符数组1必须定义得足够大,以便可以容纳得下被复制的字符串。定义字符串数组1时,其元素个数至少应该比字符串2的长度多1,应该留有存放字符串结束标志'\0'的元素。

5. 字符数组的输入输出

在程序中可以逐个输入输出字符数组元素,也可以利用系统提供的字符串处理函数,整体

输入输出字符数组中的字符串。第二章介绍的 printf、scanf、getchar、putchar、getch、getche 等库函数均可用来输入、输出字符和字符串。另外,第六章还将讲述其他的字符串处理函数,如 gets 与 puts 等函数。

(1)将字符数组中的字符串逐个字符输入输出

①在标准输入输出函数 printf 和 scanf 中使用 %c 格式说明符;

②使用 getchar、putchar、getch 和 getche 函数。

(2)字符串整体输入输出

①在标准输入输出函数 printf 和 scanf 中使用 %s 格式说明符;

②使用 gets 和 puts 函数输入、输出一行(详见第六章)。

[例4.4] 由键盘输入两个字符串,比较它们的大小。

本例完成与库函数 strcmp(详见第六章)类似的功能。C 语言中字符串比较规则与其他语言相同,即对两个字符串从左到右逐个字符(按 ASCII 码值大小)相比较,直到出现第一个不同的字符或遇到'\0'为止。若全部字符相等,则认为两字符串相等;否则比较结果以第一个不相同的字符为准。

```
main()
{
    char s1[300], s2[300];
    int result, i=0;
    printf("请输入两个字符串:");
    scanf("%s%s", s1, s2);
    while( s1[i] && s2[i] && s1[i]==s2[i] ) i++;
    result=s1[i]-s2[i];
    printf("字符串%s", s1);
    if ( result ==0 ) printf(" 等于 ");
    else if( result >0 ) printf(" 大于 ");
        else printf(" 小于 ");
    printf("字符串%s\n", s2);
}
```

运行结果:

请输入两个字符串:abcDxyz abcd

字符串 abcDxyz 小于字符串 abcd

[例4.5] 把字符串2插入到字符串1中第 i 个开始的位置上。例如:字符串1为"abcde",字符串2为"4k3y", i=3,则插入后,字符串1变成"ab4k3ycde"。

解题思路:先求出字符串2的长度 len2,然后把字符串1从最后一个字符开始后移 len2 个元素,依次移动元素,一直到第 i 个字符后移 len2 个元素,这样字符串1就留出 len2 个元素的空间,然后把字符串2插入到该空间中,并在最后加上字符串结束标志'\0',如图4.7所示。

```
#define N 100
main()
{
    char s1[N], s2[N];
```

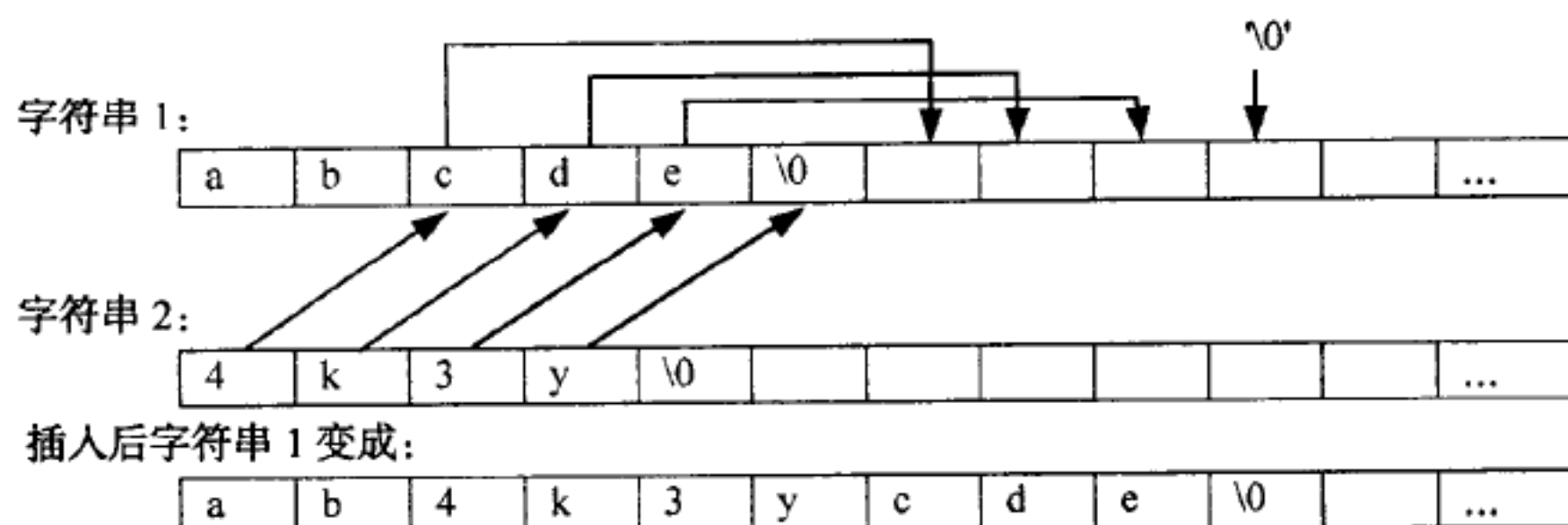


图4.7 字符串2插入到字符串1中第3个开始的位置上

```

int i, j, pos, len1=0, len2=0;
printf("请输入字符串1、字符串2以及插入位置:");
scanf("%s%s%d", s1, s2, &pos);
while( s1[len1] ) len1++; /* 求出字符串1的长度 */
while( s2[len2] ) len2++; /* 求出字符串2的长度 */
if( pos<1||pos>len1+1||len1+len2>N ) printf("不能作插入操作.\n");
else
{
    for(i=len1-1; i>=pos-1; i--)
        s1[i+len2]=s1[i]; /* s1[i]后移到 s1[i+len2]处 */
    for(i=0; i<len2; i++)
        s1[pos-1+i]=s2[i]; /* 把字符串2插入到字符串1中空出的位置中 */
    s1[len1+len2]='\0'; /* 写入字符串结束标志 */
    printf("插入后字符串1变成:%s\n", s1);
}
}

```

运行结果:

请输入字符串1、字符串2以及插入位置:abcde 4k3y 3

插入后字符串1变成:ab4k3ycde

4.1.3 二维数组

1. 二维数组的定义

二维数组的每个元素带有两个下标,在C语言中,二维数组的定义形式如下:

类型名 数组名[整型常量表达式1][整型常量表达式2];

类型名是每一个数组元素的数据类型。实际上二维数组元素组成了一个矩阵,整型常量表达式1的值就是该矩阵的行数,第一维下标值的上限为整型常量表达式1-1;整型常量表达式2的值就是该矩阵的列数,第二维下标值的上限为整型常量表达式2-1;C语言中数组下标的下限均为0,因此,二维数组元素的个数为:常量表达式1×常量表达式2。例如有下述定义:


```
int a[3][4];
```

这就定义了一个三行四列的矩阵,共12个数组元素:

	第0列	第1列	第2列	第3列
第0行	a[0][0]	a[0][1]	a[0][2]	a[0][3]
第1行	a[1][0]	a[1][1]	a[1][2]	a[1][3]
第2行	a[2][0]	a[2][1]	a[2][2]	a[2][3]

a 是二维数组名,int 是类型名,表示 a 数组的每个元素均为整型变量。每个元素有两个下标,第一个方括号中的下标代表行号,称行下标,也称为第一维下标;第二个方括号中的下标代表列号,称列下标,也称为第二维下标。例如元素 a[0][2]的行下标为0,列下标为2,它的位置在第0行,第2列。

C 语言中,可以把一个二维数组看成是一个一维数组,这个一维数组的每个元素又是一维数组。例如以上二维数组 a 可以看成是由 a[0]、a[1]、a[2]三个元素组成的一维数组;其中每个元素又是一个包含了四个整型元素的一维数组。例如 a[0]可看是由 a[0][0]、a[0][1]、a[0][2]、a[0][3]四个元素组成的一维数组。

系统为二维数组在内存中分配一块连续的存储单元,C 语言中,二维数组排列的顺序是“按行存放”的。即先存放第0行的元素,再存放第1行的元素,等等。图4.8表示了 a 数组元素在内存中的排列顺序。

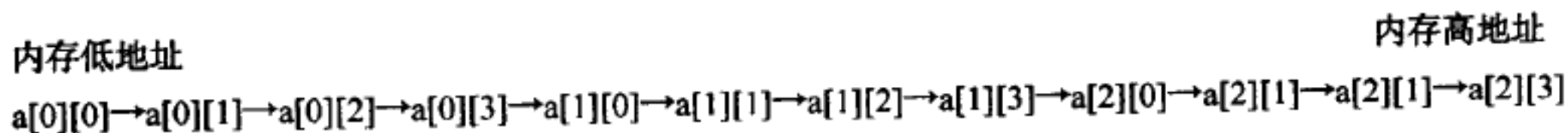


图4.8 二维数组 a 的元素在内存中的存放顺序

由二维数组的概念很容易推广到多维数组的情况,多维数组的定义形式为:

类型名 数组名[第1维长度][第2维长度]…[第 n 维长度];

多维数组的元素存放顺序是按第一维下标变化最慢,最右边的下标变化最快的原则而存放的。例如:

```
double x[2][2][3];
```

上述语句定义了一个双精度实型的三维数组 x。数组 x 共有 $2 * 2 * 3 = 12$ 个元素,其元素在内存中的存放顺序为:

x[0][0][0]→x[0][0][1]→x[0][0][2]→x[0][1][0]→x[0][1][1]→x[0][1][2]
→x[1][0][0]→x[1][0][1]→x[1][0][2]→x[1][1][0]→x[1][1][1]→x[1][1][2]

2. 二维数组元素的使用

定义了二维数组后,就可以在程序中使用数组元素,二维数组元素的使用格式如下:

数组名[下标表达式1][下标表达式2]

此处下标表达式可以是整型常量表达式或整型表达式。例如有如下定义语句:

```
int a[5][9];
```

设变量 i、j 均为整型变量,则对数组 a 的元素合法的使用形式可以是 a[0][0],a[0][1],…,a[0][8],a[1][0],…,a[4][8],a[i][j],a[i+j][i-3]等,在程序中可以像使用变量一样使用这些元素。

使用二维数组元素时应注意:

(1)所有下标表达式的值必须是整数,且不能越过数组定义时下标的上界与下界。系统也不会检查二维数组元素的下标是否越界,需要编程人员注意限制下标不要越界。

(2)C语言中,不能使用一个方括号来表示二维数组的元素,以下表示是非法的:

`a[3, 4], a[i, k]`。

3. 二维数组的初始化

可以在定义二维数组的同时给二维数组的各元素赋初值,赋初值的方法有下面几种。

(1)对全部元素赋初值

①用分行赋值方式,对全部元素赋初值

数组的每一行元素初始化赋值用花括号括起来,各元素之间用逗号“,”分开,最外层再加一对花括号。例如:

```
int a[3][5] = { {1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15} };
```

把第一个花括号内的数据赋值给第一行的元素,把第二个花括号内的数据赋值给第二行的元素,等等。这种赋值方法直观,不易遗漏,易于查错。

②按顺序赋值方式,对全部元素赋初值

二维数组存储是连续的,因此可以用一维数组赋初值的方法来给二维数组赋初值。例如:

```
int a[3][5] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
```

这与上面①中的a数组赋初值的结果相同,但比较之下这种赋值方法不够清晰。

③省略第一维的长度,但第二维长度不能省略,对全部元素赋初值

例如:

```
int a[][5] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
```

或者:

```
int a[][5] = { {1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15} };
```

这与上面①、②中a数组赋初值的结果相同。系统会根据数据的总个数分配存储空间,共15个数据,已知每行5列,得知共 $15/5=3$ 行。

(2)对部分元素赋初值

①分行赋值方式,对部分元素赋初值

可以对二维数组每一行的前面几元素赋初值。而对于后面没有赋初值的元素,系统将自动给该行后面的元素赋初值0。例如:

```
int a[4][5] = { {1, 2}, {0, 8, 9}, { }, {16} };
```

则数组a中各元素的值为:

1	2	0	0	0
0	8	9	0	0
0	0	0	0	0
16	0	0	0	0

用分行赋值方式对部分元素赋初值时,也可以省略第一维的长度。例如:

```
int a[][5] = { {1, 2}, {0, 8, 9}, { }, {16} };
```

这与语句 `int a[4][5] = { {1, 2}, {0, 8, 9}, { }, {16} };` 等价。

②按顺序赋值,对部分元素赋初值

可以按顺序赋值的方式对二维数组的部分元素赋初值。对于后面没有赋初值的元素,系统也将自动赋初值0。例如:

```
int a[3][5]={ 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

则数组 a 中各元素的值为:

1	2	3	4	5
6	7	8	9	0
0	0	0	0	0

利用这一特点,可以用如下方式:

```
int b[5][9]={0};
```

给二维数组 b 的所有元素赋初值0。

③按顺序赋值,对部分元素赋初值,并且省略第一维的长度

此时,数组第一维的大小按以下规则决定:设有 n 个初值数据,数组的第二维长度是 n2,因为是对部分元素赋初值,则 n/n2不能被整除。第一维的大小为 n/n2+1。例如:

```
int a[ ][5]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

此处共有11个初值数据,则 a 数组第一维的大小为 $11/5+1=3$ (这里11/5为整数除),即上述语句等同于 `int a[3][5]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};`。

4. 二维数组程序举例

[例4.6] 调用随机函数产生一个5行5列的二维数组(矩阵)a,要求每个数组元素均为整数,并且 $10 \leq a[i][j] \leq 99$,输出该矩阵。把矩阵 a 转置(行、列互换),然后再输出转置后的矩阵。

库函数 `randomize()` 用于初始化随机数产生函数 `random`。调用库函数 `random(n)` 产生一个0到 n-1之间的随机整数。调用这两个函数,程序中应使用库文件包含 `#include "stdlib.h"`。

把矩阵转置,即把矩阵的行、列互换,为了防止已交换过的元素再次进行交换,程序中控制列号的变量 j 是从 i+1 循环到 n-1。

程序如下:

```
#include "stdlib.h"
const int Max=99, Min=10;
main()
{
    int i, j, a[5][5], temp;
    randomize();
    for( i=0; i<5; i++)
        for( j=0; j<5; j++)
            a[i][j]=Min+random(Max-Min+1); /* 限制 Min≤a[i][j]≤Max */
    printf("随机产生的 a 矩阵如下:\n");
    for( i=0; i<5; i++)
    {
        for( j=0; j<5; j++)
            printf(" %5d", a[i][j]);
        printf("\n");
    }
    for( i=0; i<5; i++)
        for( j=i+1; j<5; j++)
```

```

    {
        temp=a[i][j];
        a[i][j]=a[j][i];
        a[j][i]=temp;
    }
    printf("a 矩阵转置后如下:\n");
    for( i=0; i<5; i++)
    {
        for( j=0; j<5; j++)
            printf( "%5d", a[i][j] );
        printf("\n");
    }
}

```

运行结果:

随机产生的 a 矩阵如下:

```

27   51   40   30   75
44   11   54   22   61
82   62   21   92   41
25   97   80   94   10
76   48   35   54   32

```

a 矩阵转置后如下:

```

27   44   82   25   76
51   11   62   97   48
40   54   21   80   35
30   22   92   94   54
75   61   41   10   32

```

[例4.7] 编写程序打印下面的杨辉三角形(输出6行)。

杨辉三角形满足以下规则:首行只有一个元素值为 1;从第二行开始首末两元素都是 1;中间的第 k 个元素等于上一行第 k-1 个元素与上一行第 k 个元素之和,如图4.9所示。

```

1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5 10 10 5  1
.
...

```

图4.9 杨辉三角形

```
#include <stdio.h>
```

```

#define N 6
main()
{
    int i, j, a[N][N];
    for ( i=0; i<N; i++)
        a[i][0]=a[i][i]=1; /* 使第一列和对角元素置1 */
    for ( i=2; i<N; i++)
        for ( j=1; j<i; j++)
            a[i][j]=a[i-1][j-1]+a[i-1][j];
    for ( i=0; i<N; i++)
    {
        for ( j=0; j<i+1; j++)
            printf( "%-5d", a[i][j]);
        printf( "\n" );
    }
}

```

运行结果:

```

1
1    1
1    2    1
1    3    3    1
1    4    6    4    1
1    5    10   10   5    1

```

4.2 结构体

4.2.1 结构体的概念

一个数组由类型相同的元素组成,但有些数据类型可能由性质不同的成员组成,这些成员相互关联组成一个整体数据。例如,一个职工的工作证号(number)、姓名(name)、性别(sex)、年龄(age)、部门(department)、工资(wage)等,这些成员都属于某个职工的数据项(如图4.10所示)。如果把 number、name、sex、age、department、wage 分别定义成相互独立的变量,则不能体现它们之间的内在联系,而在程序中应该把它们视为同一个数据类型的成员。利用 C 语言的结构体类型就可完成这一功能。例如:

number	name	sex	age	department	wage[0]	wage[1]
900806	李大磊	M	35	技术部	656.5	850.0

图4.10 某职工数据的成员


```

struct staff
{
    char number[8];
    char name[10];
    char sex;
    int age;
    char dep[20];
    float wage[2];
};

```

以上定义了一个结构体类型 `struct staff`, 它包含了 `number`、`name`、`sex`、`age`、`dep`、`wage` 等不同类型的数据项。这样就可使用 `struct staff` 类型的变量, 把一个职工的相关信息合在一起, 组成一个整体来处理。例如下面程序段:

```

struct staff worker; /* 定义一个 struct staff 类型的变量 worker */
worker.sex='M'; /* 给 worker 变量的 sex 成员赋值'M' */
strcpy(worker.name, "李大磊"); /* 给 worker 变量的 name 成员赋值"李大磊" */
printf("姓名:%s 性别:%c\n", worker.name, worker.sex); /* 输出 worker 变量的数据 */

```

4.2.2 结构体类型及结构体变量

1. 结构体类型说明的一般形式

```

struct 结构体名
{
    类型名1 结构体成员名表1;
    类型名2 结构体成员名表2;
    :
    类型名 n 结构体成员名表 n;
};

```

说明:

(1) 结构体类型名和结构体变量名由用户命名, 命名规则与标识符命名规则相同(即与变量命名规则相同)。

(2) 每个“结构体成员名表”都可以含有多个相同类型的成员名, 它们之间以逗号隔开。结构体成员的命名规则与变量名的命名规则相同。结构体成员名允许与该结构体外的变量重名, 不同结构体中的成员也可以同名, 它们代表不同的对象, 互不干扰。

(3) 结构体成员的类型可以是基本类型、数组、共用体、指针、空类型或已说明过的结构体类型等。

(4) 其中 `struct` 是 C 语言的关键字, 是结构体类型的引导字, 用于说明结构体类型以及定义结构体变量。

(5) 结构体说明的花括号后要以分号“;”结尾。

例如一本书有书号、书名、作者、价格、简介等数据项, 可以定义书的结构体类型如下:

```

struct book _type

```

```

{   int num;
    char name[60];
    char writer[30];
    float price;
    char brief[300];
};

```

结构体成员也可以是一个结构体变量,即结构体允许嵌套结构。例如:下面语句定义了一个有嵌套结构的结构体,如图4.11所示。

number	name	sex	birthday			department	wage[0]	wage[1]
			month	day	year			

图4.11 职工结构体类型的嵌套定义

```

struct date
{
    unsigned char month;
    unsigned char day;
    unsigned year;
};
struct staff
{
    char number[8];
    char name[10];
    struct date birthday;
    int age;
    char dep[20];
    float wage[2];
};

```

结构体类型说明只是说明了一个构造型数据类型,系统没有分配任何存储空间。必须定义相应结构体类型的变量,系统才为该变量分配存储空间。类型与变量是不同的概念。

2. 结构体变量的四种定义方式

可以用以下四种方式定义结构体类型的变量。

(1) 紧跟在类型说明之后定义变量

例如:

```

struct book _type
{
    int num;
    char name[60];
    char writer[30];
    float price;
    char brief[300];
};

```

```
}book1, books[3];
```

此处,在说明结构体类型 `struct book_type` 的同时,定义了一个结构体变量 `book1` 与一个结构体数组 `books`,数组 `books` 有三个元素:`books[0]`、`books[1]`、`books[2]`。

这种定义变量的一般形式为:

```
struct 结构体名
{
    结构体成员表
} 变量名表;
```

(2)先说明结构体类型,再单独进行变量定义

例如:

```
struct staff
{
    char number[8];
    char name[10];
    char sex;
    int age;
    char dep[20];
    float wage[2];
};
...
struct staff worker1, worker2;
```

此处先说明了结构体类型 `struct staff`,再由另一条语句定义变量 `worker1` 与 `worker2` 为 `struct staff` 类型的变量。

(3)说明一个无名结构体类型,直接定义变量

例如:

```
struct
{
    unsigned char month;
    unsigned char day;
    unsigned year;
}d1[5];
```

以上说明结构体类型时省略了结构体类型名,直接定义结构体数组 `d1`。若程序中不需要再次定义该结构体类型的变量,可用这种方式一次性定义变量。

(4)用 `typedef` 说明一个结构体类型名,再用类型名进行变量定义(详见本章4.5节)。

4.2.3 结构体变量的使用

结构体是一个构造型数据类型,由此定义的结构体变量的成员,也可以像其他类型的变量一样被赋值,参于表达式运算以及用于输入、输出等操作。

1. 结构体变量成员的使用

结构体变量成员的表示方式为:

结构体变量名. 成员名

运算符“.”称为结构体成员运算符,它在 C 语言中的运算优先级的级别是最高的。可以把“结构体变量名. 成员名”看成一个整体,则这个整体的数据类型与结构体中该成员的数据类型相同,可以像使用一个简单变量一样使用“结构体变量名. 成员名”。

对于多层嵌套结构体成员的使用,应按照从最外层到最内层的顺序逐层使用成员名,每层成员名之间用结构体成员运算符“.”隔开,只能对最内层的成员进行存取、运算以及输入、输出等操作。

[例4.8] 学生的数据包括学号、姓名、出生日期、三门课成绩、总分及平均分。定义一个结构变量,其中每个成员都从键盘接收数据,然后计算总分及平均分,最后输出该学生的所有数据。请注意这个例子中不同类型的结构体成员的使用方法。

```
#include "stdio.h"
#define ESC 27
struct student _type
{
    long int num;    /* 学号 */
    char name[10];  /* 姓名 */
    struct date _type
    {
        int month;
        int day;
        int year;
    } birthday;    /* 出生日期 */
    float score[3], total, average; /* 三门课成绩、总分、平均分 */
}
main()
{
    struct student _type stu;
    /* 定义 struct student _type 类型的结构体变量 stu */
    do
    {
        printf("请输入学生的学号、姓名、出生年、月、日及三门课学习成绩: \n");
        scanf("%ld%s", &stu.num, stu.name);
        /* 此处 name 是字符数组首地址,前面不要使用地址运算符"&" */
        scanf("%d%d%d", &stu.birthday.year, &stu.birthday.month,
            &stu.birthday.day);
        scanf("%f%f%f", &stu.score[0], &stu.score[1], &stu.score[2]);
        stu.total=stu.score[0]+stu.score[1]+stu.score[2];
        stu.average=stu.total/3.0;
```

```

printf("现输出该学生的数据如下:\n");
printf(" %10s%10s%16s%20s%10s%10s\n", "学号", "姓名", "出生日期",
"三门课成绩", "总成绩", "平均成绩");
printf(" %10ld%10s", stu.num, stu.name);
printf(" %6d 年%2d 月%2d 日", stu.birthday.year, stu.birthday.month,
stu.birthday.day);
printf(" %8.2f, %6.2f, %6.2f", stu.score[0], stu.score[1], stu.score[2]);
printf(" %8.2f%10.2f\n", stu.total, stu.average);
printf("按 Esc 退出,按其他键继续...\n");
} while( getch() != ESC );
/* 等待按键,按 Esc 退出,按其他键继续处理另一位学生的数据。*/
}

```

运行结果:

请输入学生的学号、姓名、出生年、月、日及三门课学习成绩:

21003040 田芷 1991 10 29 87.50 76.00 90.50

现输出该学生的数据如下:

学号	姓名	出生日期	三门课成绩	总成绩	平均成绩
21003040	田芷	1991年10月29日	87.50, 76.00, 90.50	254.00	84.67

按 Esc 退出,按其他键继续...

2. 结构体数组成员的使用

结构体数组成员的表示形式为:

结构体数组元素. 成员名

假设已定义了与[例4.8]中相同的 struct student _type 结构体类型,则可以按如下程序段来定义结构体数组和使用结构体数组成员:

```

struct student _type student[10];
student[0].birthday.year=1991; /* 对数组 student 第0个元素的出生年份赋值 */
printf(" %s\n", student[9].name); /* 输出数组 student 第9个元素的姓名 */

```

[例4.9] 从键盘上输入一段英文文字,假设每个单词由空格、制表符'\t'、换行'\n'来分隔,当输入单词 STOP 后,结束输入过程。以下程序统计这段文字中每个单词出现的次数(假设每个单词的长度不超过20个字符,不同单词总个数小于1000)。

例如,输入了一段文字:abc aa 345 abc abc 555 aa aa a 555% 555 555 STOP

统计后输出:

abc 出现3次, aa 出现3次, 345出现1次, 555出现3次, a 出现1次,

555%出现1次, STOP 出现1次,

程序中使用了库函数 strcmp(详见第六章)来比较两个字符串是否相同。程序如下:

```

#include <stdio.h>
#define N 21
#define M 1000
struct wordtype
{

```



```

    char word[N];        /* 存放单词 */
    int time;            /* 存放该单词出现的次数 */
} wd[M];
/* 上述 wd 是一个结构体数组,由于是全局变量(详见第六章),其所有成员的初
值均为0 */
main()
{
    int i, num=0;        /* num 用于统计不同单词的数目 */
    int in;              /* 变量 in 用于判定是否为已经出现过的旧单词,
                           是,则 in=1,否,则 in=0 */

    char str[N];
    printf("请输入一段英文,最后以单词 STOP 结束.\n");
    do
    {
        scanf("%s",str);
        in = 0; /* 先假设 str 为还未出现过的新单词 */
        for(i=0; i<num; i++)
            if( strcmp(wd[i].word, str) == 0)
                /* 判定 str 是否为已经出现过的旧单词 */
                {
                    wd[i].time++;
                    in = 1; /* str 是已经出现过的旧单词 */
                    break;
                }
        if ( !in ) /* 若是未曾出现过的新单词 */
        {
            strcpy(wd[i].word, str);
            wd[i].time = 1 ;
            num++;
        }
    }while( strcmp(str, "STOP") );
    printf("该段英文中,单词出现的频率统计如下:\n");
    for(i=0; i<num; i++)
    {   printf("%s 出现%d 次", wd[i].word, wd[i].time);
        if ( (i+1)%5 == 0 ) printf("\n"); /* 每输出5项结果就换行 */
    }
    printf("\n");
}

```

运行结果:

请输入一段英文,最后以单词 STOP 结束。

```
abc aa 345 abc abc 555 aa aa a 555% 555 555 STOP
```

该段英文中,单词出现的频率统计如下:

abc 出现3次,aa 出现3次,345出现1次,555出现3次,a 出现1次,
555%出现1次,STOP 出现1次,

3. 相同类型结构体变量之间的整体赋值

旧版本的 C 语言标准中,不允许对结构体变量进行整体赋值操作,而只能逐一对结构体变量中的成员进行赋值。新 ANSI C 标准增加了对结构体类型变量的整体赋值操作。例如:

```
struct
{
    float f1, f2;
} x, y;
x.f1 = 12.3;
x.f2 = 456.789;
y = x;
```

上述赋值语句 $y=x$; 相当于两条赋值语句: $y.f1=x.f1$; $y.f2=x.f2$; , 将 x 中每个成员的值都赋给 y 中对应的同名成员。这种赋值方法常用于程序中交换两个结构体变量中所有相对应的成员值。只有类型相同的结构体变量之间才能进行整体赋值。

说明:

(1) 表达式 `sizeof(结构体类型)` 或 `sizeof(结构体变量)` 的值是该结构体各成员所分配的存储空间之和。例如下面程序段(其中 `struct student _type` 类型在[例4.8]中定义):

```
struct student _type stu;
printf("%d, %d\n", sizeof(struct student _type), sizeof(stu));
```

以上程序段运行后输出:40, 40。其中成员 `num` 占用4个字节, `name` 占用10个字节, `birthday` 占用6个字节, 数组成员 `score` 占用12个字节, `total` 和 `average` 各占用4个字节, 合计共40个字节。

(2) 结构体变量名不是指向该结构的首地址, 这与数组名的含义不同。结构体变量的首地址是: `&结构体变量名`。

(3) C 语言允许结构体变量作为一个整体赋值给相同类型的变量, 但不允许把结构体变量作为一个整体进行输入、输出操作, 也不允许结构体作为一个整体参于任何表达式计算。例如下面程序段(其中 `struct student _type` 类型在[例4.8]中定义):

```
struct student _type stu;
printf("%s\n", stu); /* 非法语句 */
scanf("%d", &stu); /* 非法语句 */
printf("%ld, %s, %d, %d, %d, %f, %f, %f, %f, %f", stu); /* 非法语句 */
scanf("%ld %s %d %d %d %f %f %f %f %f", &stu); /* 非法语句 */
stu = stu + 1; /* 非法语句 */
```

以上的输入、输出操作与表达式“ $stu+1$ ”都是非法的。

4.2.4 结构体变量、结构体数组的初始化

可以在定义结构体变量或结构体数组的同时进行赋初值。

1. 结构体变量的初始化

对结构体变量进行初始化时,所赋初值顺序放在一对花括号中,系统按每个成员在结构体中的顺序一一对应赋初值,不允许跳过前边的成员给后面的成员赋初值,但可以只给前面的若干个成员赋初值,对于后面未赋初值的成员,如果是数值型和字符型数据,系统自动赋初值零。例如:

```
struct book _tp
{
    char name[60];      /* 书名 */
    char author[30];    /* 作者 */
    float price;        /* 价格 */
    struct datetp
    {
        unsigned year;
        unsigned month;
    } pubday;          /* 出版日期 */
}book1= {"SQL Server 循序渐进教程", "Petkovic", 35.0, {1999, 6} },
book2= {"VB 开发指南", "Dianne Siebold"};
printf(" %s,作者:%s,出版日期:%d 年%d 月,价格:%5.1f\n",
book1.name, book1.author, book1.pubday.year, book1.pubday.month,
book1.price);
printf(" %s,作者:%s,出版日期:%d 年%d 月,价格:%5.1f\n",
book2.name, book2.author, book2.pubday.year, book2.pubday.month,
book2.price);
```

上述程序段运行结果为:

SQL Server 循序渐进教程,作者:Petkovic,出版日期:1999年6月,价格: 35.0

VB 开发指南,作者:Dianne Siebold,出版日期:0年0月,价格: 0.0

2. 结构体数组初始化

结构体数组初始化的方法与数组的初始化相类似。由于数组中的每个元素都是一个结构体,可以将每个元素中成员的初值依次放在一对花括号内,以便区分各个元素。例如:

```
struct student _tp
{
    char num[10];
    char name[10];
    float score;
} student [3] = { { " 304001", " 张 驰", 65 }, { " 304002", " 杨 明", 78 },
{ "304005", "王丰", 86 } };
```

也可以在初始化时不指定数组的长度,系统根据所赋初值的个数决定数组元素的个数。例如下面的语句与上述语句等价:

```
struct student _tp
{
```

```

char num[10];
char name[10];
float score;
} student [ ] = { { "304001", "张驰", 65 }, { "304002", "杨明", 78 },
{ "304005", "王丰", 86 } };

```

初始化时内层的花括号可以省略,但省略后各元素的初值连成一片,容易混淆。因此,一般不省略内层的花括号。

4.2.5 位段

有时只需要1位或几位就可以存储一个信息。例如:“真”用1表示,“假”用0表示,只要1位就够存放了,如果用字符(char)类型的变量来表示,要占用内存一个字节(8位),浪费了存储空间。另外,有时要存取一个或多个字节的某几位,或对一个或多个字节的某几位进行位运算,虽然利用第二章中讲述的按位运算可以完成这些操作,但较繁琐。利用C语言提供的位段,可以解决上述问题。

1. 位段的定义

所谓位段是以位为单位定义变量占内存空间的大小。C语言中没有专门的位段类型,而是借助于结构体类型,以二进制位为单位来说明结构体中成员所占空间的大小。例如:

```

struct bit _field
{
    unsigned a : 3;
    unsigned b : 6;
    unsigned c : 11;
    int i;
} x;

```

以上定义了一个结构体变量x,它有三个位段成员和一个整型成员。系统为变量x在内存中分配的存储空间如图4.12所示。

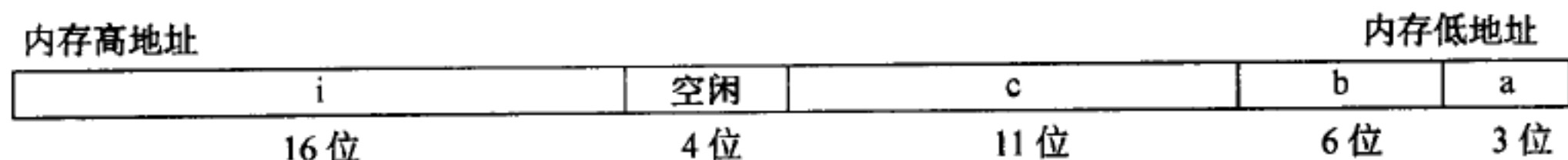


图4.12 带位段的结构体变量x的内存分配情况

位段定义的一般形式为:

```

struct [结构体名]
{
    类型名 [位段名]:整型常量表达式;
    :
} [变量名表];

```

说明:

(1) 此处类型名只能是 unsigned 或 int 类型。整型常量表达式用于指定每个位段的宽度, 即该位段占内存多少位。位段宽度的取值范围在 0~16 之间。

(2) 有时方括号内的内容可以省略。省略位段名时, 该位段称为无名位段。无名位段的作用是跳过不使用的某几个位。当无名位段宽度为 0 时, 将使下一个位段从一个新的字节开始存放。

(3) 位段成员的内存空间分配方向, 因机器而异。IBM PC 兼容机是“从低字节到高字节”分配位段成员的存储空间, 如图 4.12 所示。

(4) 不能使用数组作位段成员, 但位段变量可以是数组。

(5) 位段总长度(位数), 是各个位段成员的长度(位数)之和。位段总长度可以超过两个字节。

(6) 一个结构体内可以在定义位段成员的同时定义其他非位段成员。结构体变量的非位段成员要从一个新的字节开始分配存储空间, 中间空闲的若干位将不被使用。

2. 使用位段

位段成员的使用与结构体成员的使用相类似。

例如对于上述例子中定义的变量 x, 下面语句是合法的:

```
x.a = 7;
x.c = 300;
printf("%d", x.a + x.c >> 2);
```

说明:

(1) 位段可以进行赋值操作, 所赋之值可以是整数。赋值时, 应注意位段允许的最大值范围。例如上面 x.a 定义的位段宽度为 3, 如果 x.a = 9;, 就错了。赋值语句中, 赋值表达式的值超出位段的宽度时, 则自动取值的低位赋值。9 的二进制数是 1001, 而 x.a 的宽度只有三位, 取 1001 的低 3 位赋值给 x.a, 因此赋值后 x.a 的值为 1, 而不是 9。

(2) 位段可以按整型格式输出, 可以在 printf 函数中使用 "%d"、"%u"、"%o"、"%x" 等输出格式字符。

(3) 不能对位段成员求地址, 因此也不能由键盘读入位段值。例如: 语句 scanf("%u", &x.a); 是非法的。

(4) 位段成员可以进行算术表达式的运算, 系统自动将其转换成整型。

4.3 共用体

4.3.1 共用体的概念、类型说明和变量定义

1. 共用体的概念

程序设计有时需要在同一段内存中存取不同类型的变量, 例如, 在同一个地址开始的内存块中, 分别存取整型变量、字符型变量、实型变量的值(如图 4.13 所示)。三种变量在内存中占有不同的字节数, 但都从同一地址开始存放, 它们的值可以相互覆盖。本节将介绍利用“共用体”类型来完成这样的操作。所谓共用体类型, 就是几个不同类型的变量共占一段内存的结构。

2. 共用体类型的说明和变量定义

共用体类型说明和变量定义方式和结构体的方式完全相同。所不同的是,结构体变量中的成员各自占有自己的存储空间,而共用体变量中的所有成员占有共同的存储空间。例如:

```
union u_type
{
    char ch;
    int i;
    float f;
} v;
```

以上说明了一个共用体类型 `u_type`,如图4.13所示。同时还定义了一个该类型的共用体变量 `v`。

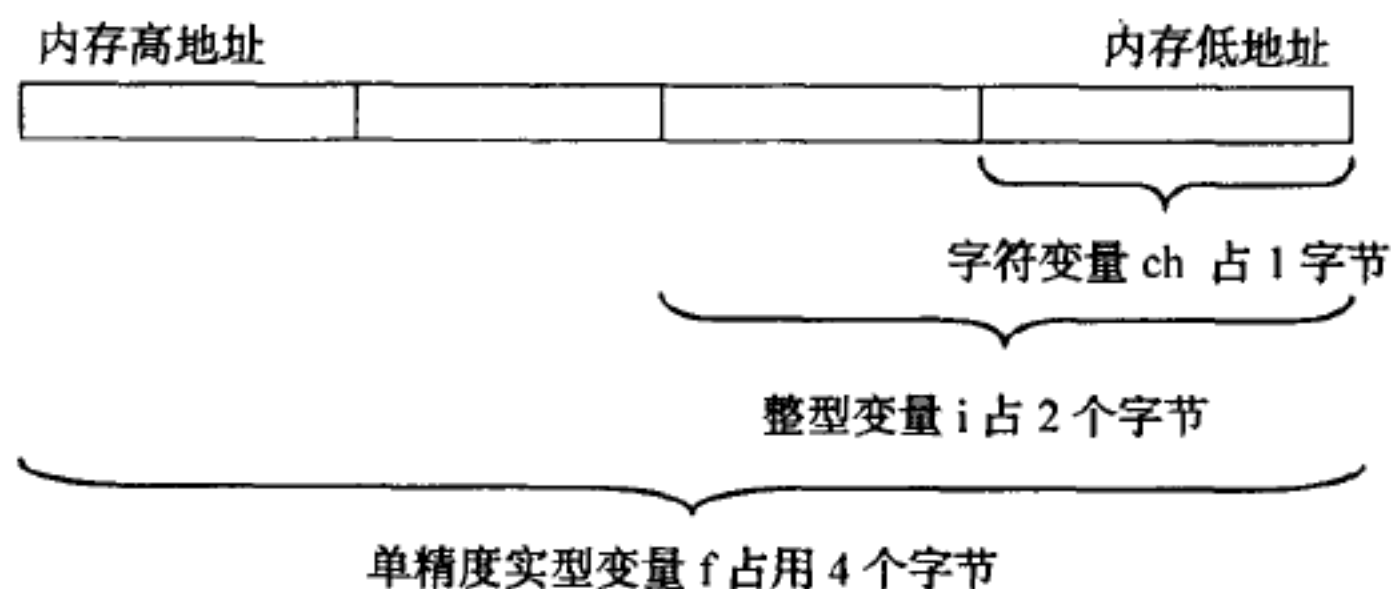


图4.13 字符变量、整型变量及实型变量共占同一段内存地址构成共用体类型

共用体类型说明的一般形式为:

```
union [共用体名]
{
    类型名1    共用体成员名1;
    类型名2    共用体成员名2;
    :
    类型名 n    共用体成员名 n;
} [共用体变量名表];
```

说明:

(1)其中 `union` 是 C 语言的关键字,用于说明共用体类型。“共用体名”、“共用体成员名”以及“共用体变量名”都是由用户定义的标识符。

(2)其中方括号内的内容可以省略,共用体变量的定义与结构体变量类似,可以在说明类型的同时定义变量;也可以先说明共用体类型,再用另一条语句定义共用体变量;还可以直接定义共用体变量,省略共用体名。

(3)共用体中的成员可以是简单变量,也可以是数组、指针、结构体和共用体(结构体的成员也可以是共用体),即共用体类型允许嵌套定义,例如:

```
union u_tp
{ struct
```

```

{
    unsigned char low, high;
}b;
unsigned w;
}r;

```

以上在共用体类型中,定义了一个结构体成员 b,如图4.14所示。本例中可以用 r.b.low 存取 r.w 的低字节的值,而用 r.b.high 存取 r.w 的高字节的值,也可以用 r.w 一次存取双字节的值。又如:

```

union u _ tp1
{
    unsigned char c[2];
    unsigned w;
} r1;

```

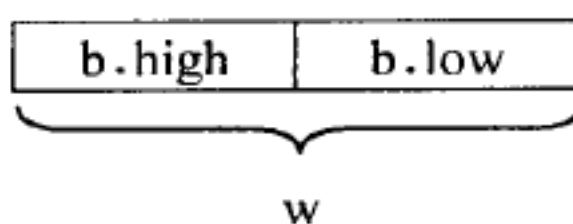


图4.14 共用体与结构体嵌套定义

本例共用体类型中,定义了一个数组成员 c,如图4.15所示。可以用 r1.c[0]存取 r1.w 低字节的值,用 r1.c[1]存取 r1.w 高字节的值。例如下列程序段:

```

r1.w=0x6141; /* 0x61是字母'a'的 ASCII 码,0x41是字母'A'的 ASCII 码 */
printf(" %c, %c\n", r1.c[1], r1.c[0]);

```

输出结果是:a, A

(4)系统为所有共用体成员分配同一地址开始的存储空间,使用覆盖的方式共享存储单元。

(5)共用体变量所占空间的大小取决于占存储空间最大的那个成员。例如上述例子中(图4.13所示),sizeof(u _ type)或sizeof(v)的值均为4,因为成员 ch、i、f 中,f 占4个字节的存储空间,是占存储空间最大的成员。

(6)由于共用体变量中所有成员共享存储空间,因此变量中的所有成员的首地址相同,而且共用体变量的地址也就是该变量成员的地址。例如上述(如图4.14所示)&r、&r.w、&r.b.low的值均相同。

3. 在定义共用体变量时赋初值

共用体变量在定义的同时只能对第一个成员的值进行初始化。例如:

```

union u _ type1
{
    int;
    float f;
} x={123};

```

以上在定义共用体变量 x 的同时给 x.i 赋初值123。

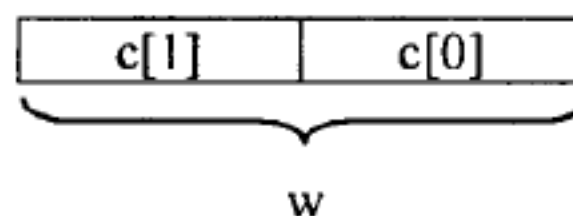


图4.15 用数组作共用体的成员

4.3.2 共用体变量的使用

1. 共用体变量的使用

共用体变量中每个成员的使用方法与结构体完全相同,其一般格式为:

共用体变量名.成员名

例如,若有定义语句:

```
union u_type
{
    char ch;
    int i;
    float f;
} v1, v2;
```

则下面均是合法的语句:

```
v1.f = 1.23;
printf("%f\n", v1.f);
scanf("%c", &v2.ch);
v1.ch = v2.ch + 32;
printf("%c\n", v1.ch);
```

说明:

(1)可以像使用简单变量一样使用“共用体变量名.成员名”。

(2)对共用体某一成员赋值,会覆盖其他成员原来的数据,原来成员的值就不存在了,因此,共用体变量中起作用的是最后一次存入的成员变量的值。例如:

```
v1.ch = 'a';
v1.i = 0x3f41;
printf("%c, %x\n", v1.ch, v1.i);
```

以上程序段中,最后一次是给共用体中整型成员变量 v1.i 赋十六进制数 0x3f41,低字节 0x41(即大写字母'A'的 ASCII 码)把前面的小写字母'a'的 ASCII 码 0x61 覆盖了,因此输出结果为:A, 3f41。

2. 共用体变量的整体赋值

新的 ANSI C 标准允许在两个类型相同的共用体变量之间进行赋值操作。例如下面的程序段:

```
v1.i = 123;
v2 = v1;
printf("%d\n", v2.i);
```

输出结果为123。

3. 应用举例

[例4.10] 某工厂的零件清单如图4.16所示。若零件是本厂生产的,则“零件来源”用车间代码(整型)表示;若零件不是本厂生产的,则“零件来源”用来源单位(字符数

零件编号	零件名称	本厂生产	零件来源
1023	JQ-123	y	3
3567	PK-789	n	ABCDcompany

图4.16 某工厂的零件清单

组)填写。要求输入、输出零件清单的数据(假设只有两个零件)。程序如下:

```
main( )
{ struct
  { char num [10];
    char name[15];
    char create;
    union
    { int no;
      char addr[20];
    } from;
  } sp[2];
  int i;
  printf("请输入零件清单:\n");
  for(i=0; i<2; i++)
  { scanf("%s%s%c", sp[i]. num, sp[i]. name, &sp[i]. create);
    if (sp[i]. create=='y' || sp[i]. create=='Y')
      scanf("%d", &sp[i]. from. no);
    else scanf("%s", sp[i]. from. addr);
  }
  printf("%10s%10s%10s%10s\n", "零件编号", "零件名称", "本厂生产", "零件来源");
  for(i=0; i<2; i++)
  { printf("%10s%10s%5c", sp[i]. num, sp[i]. name, sp[i]. create);
    if (sp[i]. create=='y' || sp[i]. create=='Y')
      printf("%15d\n", sp[i]. from. no);
    else printf("%15s\n", sp[i]. from. addr);
  }
}
```

运行结果:

请输入零件清单:

```
1023 JQ-123    y      3
3567 PK-789    n  ABCDcompany
零件编号 零件名称 本厂生产 零件来源
1023    JQ-123      y      3
3567    PK-789      n  ABCDcompany
```

上例定义了一个结构体数组 sp, 结构体定义中又包括了共用体类型, 该共用体有两个成员: 整型成员 no、字符数组成员 addr。

4.4 枚举型

实际应用中,某些变量只有几种可能的值,例如在模拟扑克牌游戏中,扑克牌只有四种花色(梅花、方块、红桃、黑桃),可以把这种变量定义成枚举类型。枚举类型就是将变量的可取值一一列举出来,变量只能存取其中之一值,存取其他值是错误的。枚举类型说明和定义变量的形式为:

```
enum [枚举类型名] { 枚举值1 [= 整型常数1], 枚举值2 [= 整型常数2],
    ..., 枚举值 n [= 整型常数 n] } [枚举型变量名表];
```

例如:enum cards {club, diamond, heart, spade} card1, card2;说明了一个 enum cards 枚举类型,同时定义了两个该类型的枚举型变量 card1 与 card2,它们的值只能取 club、diamond、heart、spade 之一。例如:card1 = club;、card2 = spade;都是合法的赋值语句。

说明:

1. enum 是 C 语言的关键字,是枚举类型的引导字,用于说明枚举类型以及定义枚举变量。例如:

```
enum weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun}; /* 说明枚举类型 */
enum colors {red, green, blue} bkcolor, tcolor; /* 说明枚举类型的同时定义枚举型变量 */
enum operator {plus, minus, times, divide}; /* 说明枚举类型 */
enum weekday workday, week _end; /* 用已定义好的枚举类型定义枚举型变量 */
```

2. 枚举类型的说明和变量定义有两种方式(如上),第一种方式把定义和说明分开;第二种方式在说明枚举类型的同时定义枚举变量。

3. 枚举类型名和枚举值均为用户定义的标识符。枚举值又称为枚举元素或枚举常量,系统将枚举值作为整型常量来处理。因为是常量,因此不能在程序执行时赋值,也不能与其他标识符同名。若定义了上述几种枚举类型,以下语句均是非法的:

```
int red;    Mon=1;    minus=2;
```

4. 方括号内的内容可以省略,如果省略“=整型常量”,编译程序按顺序给每个枚举元素一个对应的整数值,整数值从0开始,后续元素顺序加1。例如上面的例子中,Mon 的值为0,Tue 的值为1,...,Sun 的值为6。

可以在定义时指定枚举元素对应的整数值,没有指定整数值元素则在前一元素序号值的基础上顺序加1。例如:

```
enum weekday {Sun=7, Mon=1, Tue, Wed, Thu, Fri, Sat};
```

此时 Sun 的值为7,Mon 的值为1,Tue 的值为2,...,Sat 的值为6。

指定枚举元素对应的值可以是负的整数,以后的枚举元素值仍依次加1。例如:

```
enum operator {plus, minus= -3, times, divide};
```

则 plus 的值为0,minus 的值为-3,times 的值为-2,divide 的值为-1。

5. 每个枚举元素之间用逗号“,”隔开,而不是用分号“;”隔开。

6. 枚举变量只能取枚举类型说明结构中的某个枚举元素,若要对枚举型赋整数值必须进行强制类型转换。例如:

```
enum weekday {Sun=7, Mon=1, Tue, Wed, Thu, Fri, Sat}workday, week _end;
workday=Mon;
```



```
week_end=(enum weekday)(6); /* 相当于 week_end=Sat; */
```

7. 枚举变量可以进行加(减)一个整数 n 的运算。例如:

```
workday=Mon;workday++; /* 此时 workday 的值变成 Tue */
```

8. 枚举变量的值可以按其对应的整数值进行关系比较运算。例如:

```
if( workday==Mon ) ...
```

```
while( workday<Sat ) ...
```

9. 枚举变量的值可以按其对应的整数输出整数值。例如:

```
enum weekday {Sun=7, Mon=1, Tue, Wed, Thu, Fri, Sat} wd;
```

```
for( wd=Mon; wd<=Sun; wd++) printf("%d, ", wd);
```

上述程序段输出结果是:1, 2, 3, 4, 5, 6, 7, 。

4.5 typedef 的用途

C 语言中,除了可以使用数组、结构体、共用体和枚举型等构造类型之外,还可以使用关键字“typedef”对已有的类型说明一个新名称。用 typedef 说明一种新类型名的形式为:

```
typedef 类型名 新类型名;
```

上述只说明了一个数据类型的新名字,而不是产生了一种新的数据类型,原有类型名依然有效。“类型名”是在此语句之前已经定义了的类型标识符。“新类型名”是一个用户定义标识符,是新的类型名。例如:

```
typedef float REAL;
```

使用上述说明后,把标识符 REAL 说明成了一个 float 类型的类型名,REAL 就成了 float 的代名词。此后,可用标识符 REAL 来定义单精度实型变量。例如:

```
REAL x, y; /* 等价于 float x, y; */
```

习惯上将新的类型名用大写字母表示,以便与系统提供的关键字或预处理标识符区别开。typedef 可用来说明数组、结构体、共用体以及枚举型等类型名。下面举例说明。

1. typedef 用于定义数组类型名

```
typedef int ARRAY[10]; /* 说明 ARRAY 为有10个元素的整型数组类型名 */
```

```
ARRAY a, b; /* 定义 a 与 b 为整型数组变量,数组变量 a 与 b 各有10个元素 */
```

2. typedef 用于定义结构体类型名

```
typedef struct
```

```
{ char number[10];
```

```
  char name[10];
```

```
  float score[5];
```

```
} STUDENT; /* 说明 STUDENT 为一个结构体类型名 */
```

```
STUDENT stu; /* 定义 stu 为上述结构体类型的变量 */
```

3. typedef 用于定义共用体类型名

```
typedef union
```

```
{ int i;
```

```
  char ch;
```

```
} UTYPE; /* 说明 UTYPE 为一个共用体类型名 */
```

UTYPE x, y; /* 定义 x 与 y 为上述共用体类型的变量 */

4. typedef 用于定义枚举类型名

typedef enum {male, female} ETYPE; /* 说明 ETYPE 为一个枚举类型名 */

ETYPE sex; /* 定义 sex 为上述枚举类型的变量 */

说明:

(1) 综上所述, 可用如下3个步骤说明一个新类型名, 步骤④定义变量:

①先定义一个变量(如: int a[10];);

②将变量名换成新类型名(如: int ARRAY[10];);

③在最左边加上 typedef (如: typedef int ARRAY[10];), 新类型名定义完成;

④利用新类型名定义变量(如: ARRAY a, b;)。

(2) 用 typedef 说明的是类型名, typedef 不能用于定义变量。

4.6 小结

本章介绍了 C 语言构造型数据类型的概念及使用方法, 表4.1以举例的形式加以小结。

表4.1 举例构造型数据类型的类型说明、定义变量及使用变量

类型说明	变量定义	使用变量举例	说 明
	int a[10];	a[i] = 3 + a[8 - i]; scanf("%d", &a[k]); printf("%d\n", a[i]);	a 为一维数组, 有10个元素, 每个元素均为整型变量。
	char str[30];	scanf("%s", str); gets(str); strcpy(str, "abcde"); printf("%s", str); putchar(str[10 - i]);	str 为字符数组, 共30个元素, 每个元素均为字符型变量, str 中可存放一个长度 ≤ 29 的字符串。
	float f[][3] = {{1}, {10}};	f[i][j+1] = 3 * i + j; scanf("%f", &f[1][1]); printf("%f", f[i][j]);	f 为二维数组, 有6个元素, 相当于一个2行3列的矩阵, 每个元素均为单精度实型变量。
struct stutp { char name[10]; float score[3]; float aver, total; } s;	struct stutp st[15];	scanf("%f", &s.score[0]); scanf("%s", st[5].name); s.aver = s.total / 3.0; printf("%f", st[14].total);	说明了一个结构体类型 stutp, 并且定义了一个结构体变量 s 与一个结构体数组 st。
struct fieldtp { unsigned a : 3; unsigned : 2; unsigned b : 7; };	struct fieldtp f1, f2[10];	f1.b = 19; i = f2[1].b >> 3; printf("%d", f1.b);	说明了一个位段类型 fieldtp, 其中有一个无名位段, 并定义一个位段变量 f1 与一个位段数组 f2。
union au { unsigned a[2]; unsigned long b; } u1[8];	union au u2;	u1[6].a[0] = 0x3a4f; scanf("%u", &u2.b); printf("%u", u2.a[1]);	说明了一个共用体类型 au; 定义了一个共用体变量 u2 与一个共用体数组 u1。
typedef int AR[10];	AR x;	scanf("%d", &x[9]); x[i] = x[j+1] / 2;	用 typedef 说明了一个新类型名 AR, 用 AR 定义了一维数组变量 x, x 有10个元素, 每个元素均为整型变量。

习 题

一、选择题(每题只有一个正确答案)

4.1 C 语言中数组元素下标的数据类型是【1】。

- 【1】 A) 整型常量 B) 整型表达式
C) 任何类型的表达式 D) 整型常量或整型表达式

4.2 C 语言中, 定义一维数组元素的形式是: 类型名 数组名【2】。

- 【2】 A) [整型常量] B) [整型表达式]
C) [整型常量表达式] D) [整型常量或整型表达式]

4.3 以下在定义一维数组 a 的同时, 给 a 数组所有元素赋初值 0, 正确的语句是【3】。

- 【3】 A) `int a[5] = {0};` B) `int a[5] = (0, 0, 0, 0, 0);`
C) `int a[5] = { };` D) `int a[5] = {5, 0};`

4.4 若有定义语句 `static int a [10];`, 则数组 a 中所有元素【4】。

- 【4】 A) 在程序运行阶段被赋初值 0 B) 在程序编译阶段被赋初值 0
C) 没有确定的初值 D) 在程序运行或编译阶段被赋初值 0

4.5 下面程序段的运行结果是【5】。

```
int i=1, a[ ]={ 1, 5, 10, 9, 13, 7 };
while( a[i]<=10 ) a[i++] += 2;
for( i=0; i<6; i++ ) printf( "%d ", a[i] );
```

- 【5】 A) 2 7 12 11 13 9 B) 1 7 12 11 13 7
C) 1 7 12 11 13 9 D) 1 7 12 9 13 7

4.6 以下对 C 语言字符数组的错误描述是【6】。

- 【6】 A) 字符数组可以存放字符串
B) 字符数组中的字符串可以整体输入、输出
C) 可以在赋值语句中通过赋值运算符“=”对字符数组整体赋值
D) 字符数组中字符串的结束标志是‘\0’

4.7 以下语句把字符串“abcde”赋初值给字符数组, 不正确的语句是【7】。

- 【7】 A) `char s[] = "abcde";` B) `char s[] = {'a', 'b', 'c', 'd', 'e', '\0'};`
C) `char s[] = { "abcde" };` D) `char s[5] = "abcde";`

4.8 若有定义语句 `char s1[] = "abc", s2[9], s3[] = "ABCD", s4[] = {'a', 'b', 'c'};`, 则对库函数 `strcpy` 不正确的调用是【8】。

- 【8】 A) `strcpy(str1, "Ok!");` B) `strcpy(str2, "Ok!");`
C) `strcpy(str3, "Ok!");` D) `strcpy(str4, "Ok!");`

4.9 下面程序段的输出结果是【9】。

```
char s1[ ] = "ABCDEF", s2[ ] = "abc";
int i;
strcpy(s1, s2);
for( i=0; i<6; i++ )
    if( s1[i] ) printf("%c", s1[i]);
```

【9】 A)abcEF B)ABCDEF C)abcDEF D)abc

4.10 以下程序的输出结果是【10】。

```
main( )
{
    char str[100]="Current date is Wed 12-30-2099.";
    int i,j;
    for(i=0, j=0; str[i]; i++)
        if( !(str[i]>='A' && str[i]<='Z' || str[i]>='a' && str[i]<='z') )
            str[j++] = str[i];
    str[j] = '\0';
    printf("%s\n", str);
}
```

【10】 A)12-30-2099. B)12302099
C)Current date is Wed D)Current date is Wed 12-30-2099.

4.11 以下不能对二维数组 a 进行正确的初始化的语句是【11】。

【11】 A)int a[2][3]={0};
B)int a[][3]={1, 2, 3, 4, 5};
C)int a[2][3]={ {1, 2}, {3, 4}, {5, 6} };
D)int [2][3]={ {1}, {3, 4, 5} };

4.12 下面程序段的输出结果为【12】。

```
int a[][3]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
printf("%d\n", a[2][1]);
```

【12】 A)2 B)4 C)7 D)8

4.13 下面程序段的输出结果是【13】。

```
int m[3][3] = { { 1 }, { 2 }, { 3 } };
static int n[3][3] = { 1, 2, 3 };
printf("%d\n", m[2][0] + n[0][2]);
```

【13】 A)0 B)4 C)6 D)3

4.14 下面程序段的运行结果是【14】。

```
int i, x[3][3]={ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
for( i=0; i<3; i++ ) printf("%d", x[i][2-i]);
```

【14】 A)369 B)963 C)753 D)357

4.15 若 a[][3]={ 1, 2, 3, 4, 5, 6, 7 };, 则 a 数组第一维的大小是【15】。

【15】 A)2 B)3 C)4 D)无确定值

4.16 下面程序段的运行结果是【16】。

```
int a[2][3]={ 1, 2, 3 }, i, j;
for(i=0; i<2; i++)
    for(j=0; j<3; j++)
        { a[i][j] = a[i*j%2][j] + a[i][(i+j)%3];
          printf("%d, ", a[i][j]);
        }
```

- 【16】A) 2, 4, 6, 2, 0, 8, B) 1, 2, 3, 0, 0, 0,
 C) 1, 2, 3, 2, 0, 6, D) 2, 4, 6, 2, 0, 6

4.17 定义一个结构体变量时,系统分配给它的内存大小是【17】。

- 【17】A) 各成员所需内存量的总和
 B) 成员中占内存量最大者所需的容量
 C) 结构中第一个成员所需内存容量
 D) 结构中最后一个成员所需内存容量

4.18 在 C 程序中,使用结构体的目的是【18】。

- 【18】A) 将一组相关的数据作为一个整体,以便程序使用
 B) 将一组相同数据类型的数据作为一个整体,以便程序使用
 C) 将一组数据作为一个整体,以便其中的成员共享存储空间
 D) 将一组数值一一列举出来,该类型变量的值只限于列举的数值范围内

4.19 若有如下定义,则正确的赋值语句为【19】。

```
struct date2
{
    long i;
    char c;
}two;
struct date1
{
    int cat;
    struct date2 three;
}one;
```

- 【19】A) one.three.c='A'; B) one.two.three.c='A';
 C) three.c='A'; D) one.c='A';

4.20 以下对 C 语言共用体类型数据的描述中,不正确的是【20】。

- 【20】A) 共用体变量占的内存大小等于最大的成员的容量
 B) 共用体类型可以出现在结构体类型定义中
 C) 共用体变量不能在定义时初始化
 D) 同一共用体中各成员的首地址相同

4.21 下面程序段的输出结果为【21】。

```
struct date
{
    int a;
    char s[5];
}arg={27, "abcd"};
arg.a -= 5;
strcpy(arg.s, "ABCD");
printf("%d, %s\n", arg.a, arg.s);
```

- 【21】A) 22, ABCD B) 27, abcd C) 22, abcd D) 27, ABCD

4.22 下面程序段运行结果是【22】。

```
struct st _type
{
    char name[10];
```



```

    float score[3];
};
union u _ type
{
    int i;
    unsigned char ch;
    struct st _ type student;
} t;
printf("%d\n", sizeof(t));

```

【22】A)25 B)12 C)3 D)22

4.23 下面程序段的运行结果是【23】。

```

enum weekday { aa, bb=2, cc, dd, ee } week=ee;
printf("%d\n", week);

```

【23】A)4 B)5 C)ee D)0

4.24 以下对枚举类型名的定义中正确的是【24】。

【24】A)enum a={sum, mon, tue}; B)enum a {sum=9, mon=-1, tue};
 C)enum a={"sum", "mon", "tue"}; D)enum a {"sum", "mon", "tue"};

4.25 下列关于 typedef 语句的描述,错误的是【25】。

【25】A)用 typedef 只是对原有的类型起个新名,并没有生成新的数据类型
 B)typedef 可以用于变量的定义
 C)typedef 定义类型名可嵌套定义
 D)利用 typedef 定义类型名可以增加程序的可读性

4.26 若 typedef char STRING[255]; STRING s;,则 s 是【26】。

【26】A)字符指针数组变量 B)字符数组变量
 C)字符变量 D)字符指针变量

二、填空题

4.27 下面程序段在 w 数组中插入一元素 x,w 数组中的数已按由小到大顺序存放,插入前数组有 n 个元素。要求插入 x 后数组中的数仍有序(由小到大顺序存放)。

```

int w[15]={1, 3, 3, 6, 9, 15}, n=6, x=8, i, p=0;
while( x > w[p] ) p++; /* 找出插入的位置 */
for( i=n; i>p; i-- ) w[i]=【1】;
【2】=x;
n++; /* 插入后元素个数加 1 */

```

4.28 下面程序段的运行结果是【3】。

```

int a[ ]={ 5, 1, 15, 9, 6 }, i, j, temp;
for( i=1; i<5; i++)
{
    temp=a[i];
    j=i-1;
    while( j>=0 && temp>a[j] ) a[j+1]=a[j--];
    a[j+1]=temp;
}

```

```
for( i=0; i<5; i++ ) printf( " %d, ", a[i] );
```

- 4.29 有 n 个数分别存放在数组 $a[0] \sim a[n-1]$ 中。下面程序段的功能是从 a 数组中查找值为 y 的元素；若找到，则输出该元素的下标值，找不到，则输出 -1 。其中 k 为整型变量。

```
k=n-1;
while( k>=0 &&【4】) k--;
printf(" %d\n", k);
```

- 4.30 下面程序段的功能是输出字符数组 s 中存放的字符串的长度， len 是整型变量。

```
len = 0;
while (【5】) len++;
printf(" %d\n", len);
```

- 4.31 下面程序段的输出结果是【6】。

```
union example
{ struct
  { int x, y;
  } in;
  int a;
  int b;
}e;
e.a=1; e.b=2;
e.in.x = e.a * e.b;
e.in.y = e.a + e.b;
printf(" %d, %d", e.in.x, e.in.y);
```

- 4.32 下面程序是将原码转换成密码，其他字符不变。码表为：

原码 'a' 'b' 'z' 'd'

密码 'd' 'z' 'a' 'b'

例如，原文为 `abort,zap123`，则密文为 `dzort,adp123`。

```
#include <stdio.h>
main()
{ char s[20]="abort,zap123", t[20], tab[4][2]={{'a','d'},{'b','z'},{'z','a'},
{'d','b'}};
  int i, j=0;
  strcpy(t,s);
  while ( t[j] )
  { for (i=0; i<4; i++)
    if ( t[j] == 【7】 )
    { t[j] = tab[i][1];
      break;
    }
    【8】;
  }
```

```
    printf(" %s : %s\n",s,t);
}
```

- 4.33 下面程序计算2000年某月某日是星期几(已知2000年元旦是星期六)。

```
#include <stdio.h>
main ()
{   int daytab[12]={ 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    【9】 weekname[][10]={"Sun", "Mon", "Tues", "Wednes", "Thurs", "Fri", "Satur"};
    int i, m, d, week, days;
    printf("Input date:month=?,day=?\n");
    scanf(" %d %d", &m, &d);
    if( (m<=0||m>12)|| (d>【10】||d<=0) ) printf("Date Error\n");
    else
    {   days=d;
        for(i=0;i<m-1;i++)
            days+=daytab[i];
        week=(days+5)%【11】;
        printf("It's %sday\n", weekname[week]);
    }
}
```

- 4.34 下面程序用来检查二维数组是否对称(即:对所有 i,j 都有 $a[i][j]=a[j][i]$)。

```
main()
{   int a[4][4]={ 1,2,3,4, 2,2,5,6, 3,5,3,7, 8,6,7,4 };
    int i,j,found=0;
    for( j=0; j<4; j++ )
    {   if( found ) break;
        for( i=0; i<4; i++ )
            if( 【12】 )
            {   found=【13】;
                break;
            }
    }
    if( found ) printf( "不对称\n" );
    else printf( "对称\n" );
}
```

- 4.35 下面程序的运行结果是【14】。

```
main()
{   enum em {em1=3, em2=1, em3};
    char *aa[]={ "AA", "BB", "CC", "DD" };
    printf(" %s %s %s\n", aa[em1], aa[em2], aa[em3]);
}
```

}

三、程序设计题

- 4.36 输入 10 个整数存入数组中,找出其中最小数和次最小数。
- 4.37 编写程序,交换数组 a 和数组 b 中的对应元素。
- 4.38 有 10 个数围成一圈,求出相邻三个数之和的最小值。
- 4.39 有 5 个候选人,候选人编号从 1 到 5。30 个选民,每个选民可选一个候选人编号,也可选 0 表示投弃权票。统计输出每个候选人所得票数以及弃权票数。
- 4.40 有 1000 个编了号的产品,编号从 1001 到 2000,从中随机抽取 10 个产品来检验。编写程序,由小到大输出被抽取的产品编号。
- 4.41 对数组 A 中的 $N(0 < N < 100)$ 个整数从小到大进行连续编号,要求不能改变数组 A 中元素的顺序,且相同的整数具有相同的编号。例如:
若 A 数组为 {5,3,4,7,3,5,6},则输出为: 3,1,2,5,1,3,4。
- 4.42 输入若干个整数(少于 50 个),以 -1 结束输入,把这些数存入数组 a 中,并输出。另外,找出 a 数组中的所有素数存入数组 b,并按每行 5 个元素的格式由大到小输出这些素数。
- 4.43 某国王问棋艺高超的宰相:“欲朕赐何物?”宰相曰:“只需在棋盘的第一格中放 1 粒谷,第二格中放 2 粒,如此每往下一格,谷粒加倍,望陛下将第 64 格中的谷粒恩赐微臣。”国王欣然允诺。试问贪婪的宰相欲得谷粒几何?编程计算之。(共 9223372036854775808 粒)
- 4.44 编写程序实现将字符串 str2 拷贝到字符串 str1。
- 4.45 从键盘输入一个字符串,将该字符串中的字符排序。然后再输入一个字符,并用折半查找法找出该字符在已排序的字符串中的位置。若该字符不在字符串中,则输出提示信息。
- 4.46 编写程序,将一个字符串反向存放。
- 4.47 编写程序,输入两个字符串 str 与 substr,删除主字符串 str 中的所有子字符串 substr。
- 4.48 输入一串英文文字,统计其中每个字母(不区分大小写)的数目。
- 4.49 从键盘输入两个字符串 s1 与 s2,并在 s1 串中的最大字符后边插入字符串 s2。
- 4.50 把 1 到 25 的自然数按行顺序存入一个 5×5 的二维数组中,然后输出该数组的右上半三角。
- 4.51 输入一个 3×4 的实数矩阵 a,和一个 4×2 的实数矩阵 b,计算两矩阵的积 $c = a \times b$,c 中各元素保留 2 位小数,第 3 位小数四舍五入。
- 4.52 找出一个二维数组中的鞍点,即该位置上的元素在该行上最大,在该列上最小。有可能没有鞍点。
- 4.53 键盘输入一正整数 n ($1 \leq n \leq 20$),打印 $n \times n$ 阶右手旋转方阵。例如,若 $n = 4$,则输出:
- | | | | |
|---|----|----|----|
| 1 | 12 | 11 | 10 |
| 2 | 13 | 16 | 9 |
| 3 | 14 | 15 | 8 |
| 4 | 5 | 6 | 7 |
- 4.54 利用结构体类型编制一程序,实现输入三个学生的学号,数学、语文、英语成绩,然后计算每位学生的总成绩以及平均成绩并按总分由大到小输出成绩表。
- 4.55 定义一个包括年、月、日成员的结构体变量,将其转换成这一年的第几天并输出。应注意

闰年的二月有29天,表达式“(year%4==0&&year%100!=0)|| (year%400)==0”值为真,即为闰年,其中 year 表示年号。

- 4.56 有一个 unsigned long 类型整数,分别将其前2个字节和后2个字节作为两个 unsigned int 类型整数输出(设一个 int 型数据占2个字节)。
- 4.57 定义枚举类型 money,用枚举元素代表人民币的面值。人民币面值包括1、2、5分,1、2、5角,1、2、5、10、20、50、100元。

第 5 章

指 针

指针是 C 语言的精华,是 C 语言最重要的内容之一,也是最难掌握的部分。在程序中使用指针来处理数据、变量、数组、字符串、函数、结构体、文件及动态分配内存等。正确地使用指针,可以使程序精简、灵活、高效。指针的概念比较复杂,如果误用指针,程序运行将出现意想不到的错误,这是初学指针时应注意的问题。因此指针是 C 语言的重点和难点。

本章讲述的主要内容包括:变量的地址和变量的值、指针的基本概念;指针变量的定义、赋值;指针基本运算(包括取地址运算、存取指针所指的内容、移动指针、指针相减等);指针与一维数组的关系、数组名与地址的关系、使用指针存取数组元素;使用指针处理字符串;二维数组与指针的关系、二维数组与数组指针的关系;指向指针的指针的概念;使用指针变量存取结构体变量成员数据、指针与共用体变量的关系以及指针与枚举型变量的关系。

建议本章授课 8 学时,上机 6 学时,自学 10 学时。

5.1 指针与指针变量

5.1.1 指针的基本概念

在第二章的 2.1.3 节中讲述了内存地址的概念,计算机内存是由一片连续的存储单元组成,操作系统给每个单元一个编号,这个编号称为内存单元的地址(简称地址),每个存储单元占内存一个字节。定义一个变量,不仅规定了该变量的类型、名称,而且还在内存中为该变量分配了一块(连续的)存储单元,我们把这块存储单元的首地址称为该变量的指针(也称为变量的地址)。例如定义变量 `float f;`,假设系统为变量 `f` 分配到地址为 1500、1501、1502、1503 的 4 个连续存储单元,则地址 1500 就称为变量 `f` 的指针(也称变量 `f` 的地址是 1500)。因此变量的地址就是变量的指针。变量的值和变量的地址是不同的概念,变量的值是该变量在内存单元中的数据。

下面用一个卡片管理系统来说明如何存取变量的值。

一个卡片系统由许多张卡片组成。要快速存取每张卡片上的数据,需要给每张卡片编号(标明每张卡片的地址);为了方便存取卡片上的数据,便于人们记忆,还需要给每张卡片取一个名字,并把名字与编号一一对应起来。这样卡片有了编号和名字,就能高效、方便地管理整个卡片系统了。如图 5.1 所示,可以通过卡片的名字存取卡片上的数据,例如只要指出卡片的名字 `f`,就可以取出与之相对应的编号 1500 处的数据 456.789。有一类很特殊的卡片,它的内容专门存放其他卡片的编号(地址)。如图 5.1 中,卡片 `p` 上的数据是 1500,是卡片 `f` 的编号。通

过卡片 p 也可以间接地存取卡片 f 的数据,先取到卡片 p 上的数据 1500,再到 1500 编号处存取数据。

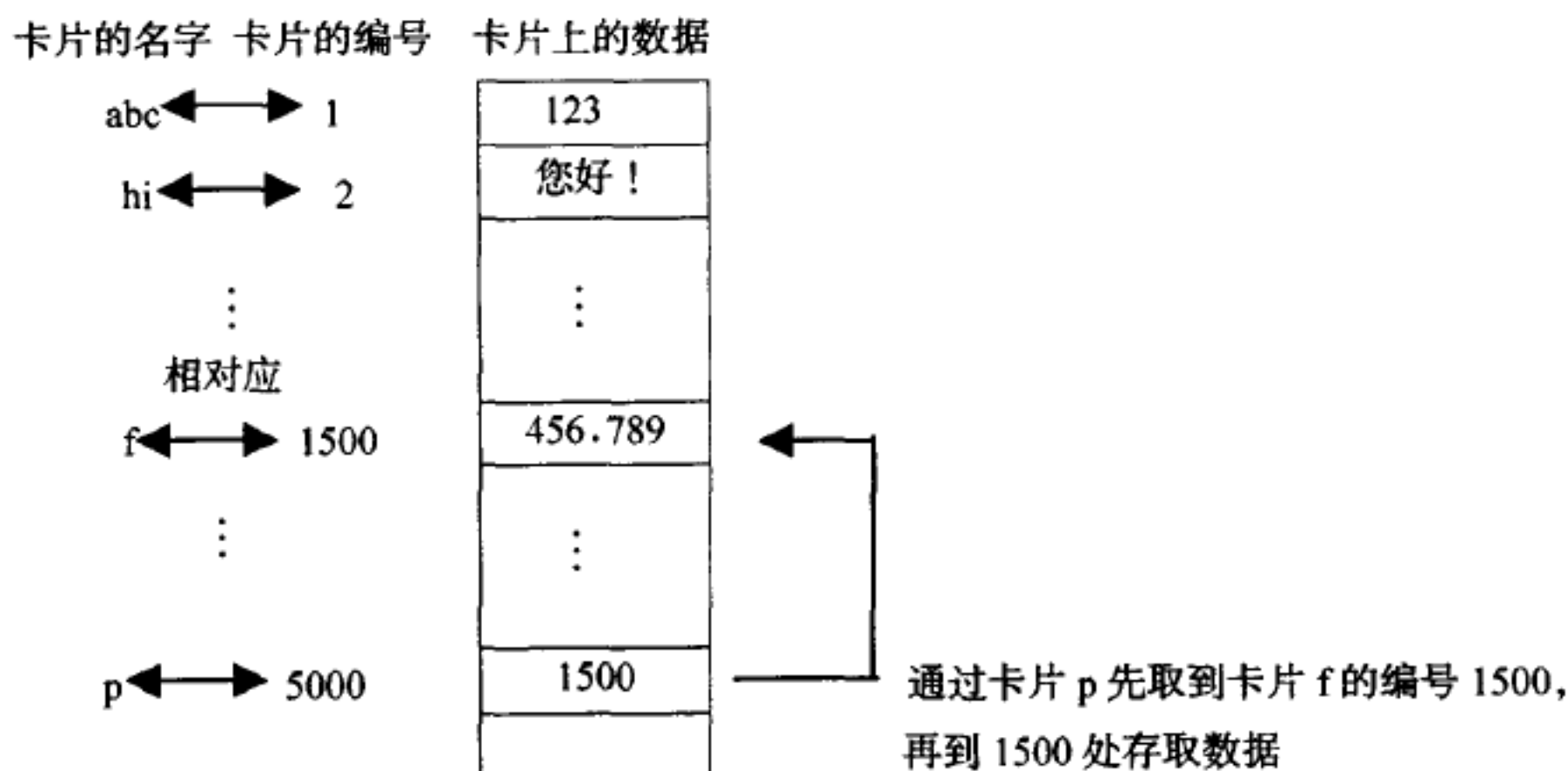


图 5.1 一个卡片管理系统

变量值的存取与卡片数据的存取极为相似。每个变量都有名字和地址。程序员在编写程序时,使用变量名来存取该变量的值。与之相对应,计算机系统使用变量的地址来存取该变量的值。这是因为程序经过编译系统编译后,每个变量都分配了相应的存储单元,每个变量名都变成了与之相对应的地址。假设 float 类型的变量 f 的地址是 1500,若执行语句 `f=456.789;`,计算机系统就把 456.789 存入到地址为 1500 开始的四个存储单元中(单精度实型占内存 4 字节);若执行语句 `printf("%f\n",f);`,则从地址 1500 开始的四个存储单元中取出数据 456.789 输出到屏幕。对于计算机而言,变量名都被翻译成了地址,系统按变量地址存取变量的值,这称为“直接存取”方式。

有一类特殊的变量,它的内容专门存放其他变量的地址,这样的变量称为指针变量。如图 5.2 中指针变量 p 的值为 1500,是变量 f 的地址。通过变量 p 可以间接地存取变量 f 的数据,先取出 p 的值“地址 1500”,再到“地址 1500”处存取数据,这种通过变量 p 间接得到变量 f 的地址,然后再存取变量 f 的值的方式称为“间接存取”方式,见图 5.2。

用来存放指针(地址)的变量就称作指针变量。上述变量 p 就是一个指针变量。若把变量 f 的地址赋值给指针变量 p,则称指针变量 p 指向变量 f(见图 5.3)。

5.1.2 指针变量的定义

定义指针变量的一般形式为:

类型名 * 标识符;

其中“标识符”是指针变量名,标识符前加“*”号表示该变量是指针变量,用于存取放地址,“类型名”表示该指针变量所指向变量的类型。例如:

```
int i, *ip, *jp; /* 定义 ip 与 jp 两个指针变量,它们可以指向 int 类型的变量 */
float f, *p;    /* 定义 p 为指针变量,它可以指向 float 类型的变量 */
```

若 `ip=&i;`,则称指针变量 ip 指向整型变量 i,因为把变量 i 的地址赋值给了 ip。赋值语句 `ip=&f;` 是非法的,一个指针变量只能指向类型相同的变量,一个指向整型变量的指针变量不

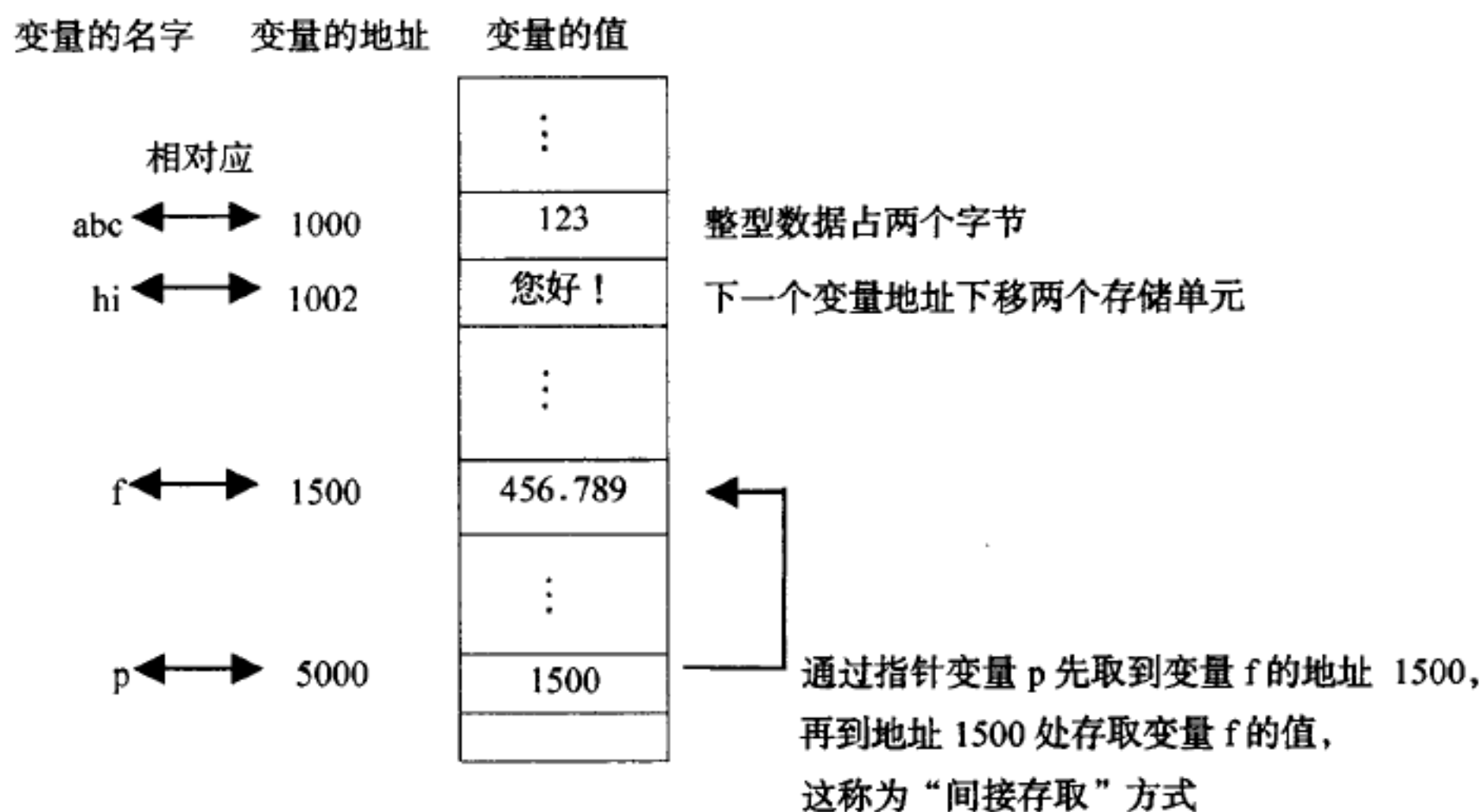


图 5.2 变量在内存中的存储分配及变量值的存取

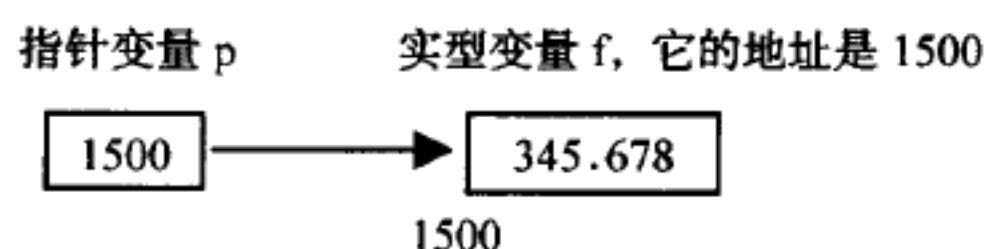


图 5.3 指针变量 p 指向变量 f

允许指向实型变量。

5.1.3 指针变量的赋值

指针变量可以通过不同的方法获得一个地址值。

1. 通过地址运算“&”赋值

地址运算符“&”是单目运算符,运算对象放在地址运算符“&”的右边,用于求出运算对象的地址。通过地址运算“&”可以把一个变量的地址赋给指针变量。若有语句:

```
float f, *p;
p=&f;
```

执行后把变量 f 的地址赋值给指针变量 p,指针变量 p 就指向了变量 f。

2. 指针变量的初始化

与动态变量的初值一样,在定义了一个(动态的)指针变量之后,其初值也是一个不确定的值。可以在定义变量时给指针变量赋初值,如 `float f, *p=&f;`,则把变量 f 的地址赋值给指针变量 p,此语句相当于 `float f, *p; p=&f;` 这两条语句。

3. 通过其他指针变量赋值

可以通过赋值运算符,把一个指针变量的地址值赋给另一个指针变量,这样两个指针变量均指向同一地址。例如,有如下程序段:

```
int i, *p1=&i, *p2;
```

```
p2=p1;
```

执行后指针变量 p1 与 p2 都指向整型变量 i。

注意,当把一个指针变量的地址值赋给另一个指针变量时,赋值号两边指针变量所指的数据类型必须相同。例如:

```
int i, *pi=&i;
float *pf;
```

则语句 pf=pi; 是非法的,因为 pf 只能指向实型变量,而不能指向整型变量。

4. 用 NULL 给指针变量赋空值

除了给指针变量赋地址值外,还可以给指针变量赋空值,如:

```
p=NULL;
```

NULL 是在 stdio.h 头文件中定义的预定义标识符,因此在使用 NULL 时,应该在程序中加上文件包含 #include "stdio.h"。在 stdio.h 头文件中 NULL 被定义成符号常量,与整数 0 对应。执行以上的赋值语句后, p 为空指针,在 C 语言中当指针值为 NULL 时,指针不指向任何有效数据,因此在程序中为了防止错误地使用指针来存取数据,常常在指针未使用之前,先赋初值为 NULL。由于 NULL 与整数 0 相对应,所以下面三条语句等价:

```
p=NULL; 或 p=0; 或 p='\0';
```

但通常都使用 p=NULL; 的形式,因为这条语句的可读性好。NULL 可以赋值给指向任何类型的指针变量。

5. 通过调用标准函数赋值

可以调用库函数 malloc 和 calloc 在内存中开辟动态存储单元,并把所开辟的动态存储单元的首地址赋给指针变量。由于这两个函数返回的是“void *”无类型指针类型,因此将它们的返回值赋值给指针变量时要进行强制类型转换。

(1) malloc 函数

调用该函数,在内存的动态存储区中分配一个指定长度(以字节为单位)的连续存储空间,如果调用该函数成功,则返回所分配空间的首地址,如果分配不成功,则返回 NULL。调用形式为:

```
malloc(表达式)
```

其中表达式的值表示所要分配存储空间的字节数。例如:

```
int *p;
p=(int *)malloc(sizeof(10 * int));
```

执行上述程序段,将在内存的动态存储区中分配 sizeof(10 * int) 即 20 个字节的连续存储单元块,并把这块连续存储单元的首地址赋值给指针变量 p。其中用了 (int *) 强制类型转换,将 (void *) 类型转换成 (int *) 类型。

(2) calloc 函数

调用该函数,在内存动态存储区中分配一块连续的存储空间。函数调用形式为:

```
calloc (n, size)
```

分配 n 个长度为 size(以字节为单位)的连续空间。如果调用该函数成功,则返回所分配空间的首地址,如果分配不成功,则返回 NULL。

(3) free 函数

调用该函数,用来释放由 malloc 或 calloc 函数分配的存储空间,调用该函数不返回任何

值。函数调用形式为：

```
free(指针变量)
```

其中指针变量应指向调用 malloc 或 calloc 时所分配存储区的首地址,如 free(p);。在程序设计过程中,若动态分配的空间不再使用时(如函数调用结束),应及时释放所分配的内存空间。

有关指针变量的赋值,不仅仅是上述 5 种,指针变量还可以指向数组、字符串、结构体、函数、文件,本章以下的章节将讲述前三种概念,在第六章涉及指针与函数,第八章涉及指针与文件。

5.2 指针运算符

5.2.1 指针运算符

在 5.1.1 节描述了怎样用指针变量间接存取变量的数据,本节讲述具体实现方法。为了使用指针变量间接存取变量中的数据,C 语言中提供了指针运算符“*”。指针运算符是单目运算符,运算对象必须放在指针运算符“*”右侧,而且运算对象只能是指针变量或地址,可以用“指针运算符”来存取相应的存储单元中的数据。例如,有下面的程序段:

```
int i=123,j,*p;
p=&i;
```

则 $j=*p$; 和 $j=* \&i$; 都将把变量 i 的值赋给变量 j。而 $*p=456$; 和 $* \&i=456$; 都将把整数 456 赋给变量 i。此处表达式 $*p$ 和 $* \&i$ 都代表变量 i。运算符 & 和 * 都是单目运算符,它们具有相同的优先级,结合方向均为“从右到左”,因此 $* \&i$ 相当于 $*(\&i)$,表达式 $\&i$ 求出变量 i 的地址。由此得出结论:

1. 表达式 $* \&i$ 等价于变量 i;
2. 当一个指针变量 p 指向某变量 i 时($p=\&i$),则表达式 $*p$ 与变量 i 等价。例如:

```
p=&i;
printf("%d\n", *p)    /* 相当于 printf("%d\n", i); */
j=(*p)++;             /* 相当于 j=i++; */
```

注意表达式 $(*p)++$ 中的括号不能省略,如果没有括号,由于“*”与“++”均为单目运算符,优先级相同,结合方向是“从右到左”,因此 $*p++$ 等价于 $*(p++)$,此时先使用变量 i 的值,再使 p 的值改变,这样 p 就不再指向变量 i 了。

指针变量的值是可以改变的,一个指针变量在某一时刻指向变量 i,在另一时刻可以指向变量 j。例如:

```
int i=1, j=2, *p;
p=&i;
*p +=100; /* 相当于 i+=100; */
p=&j;
(*p)--;   /* 相当于 j--; */
```

通过 $*p$ 方式存取它所指向变量的值是以间接存取的形式进行的,所以比直接存取一个

变量要多费时间。例如执行上述语句 $(*p) += 100$; 比执行 $i += 100$; 费时多。但由于 p 是变量, 我们可以通过改变它的指向, 间接存取不同变量的值, 这给程序设计带来灵活性, 也使程序代码编写更为简洁、高效。

取地址运算符“&”与指针运算符“*”作用在一起时, 有相互“抵消”的作用。例如若有 $\text{int } i, *p = \&i$; 则:

1. $*p \xleftrightarrow{\text{相互等价}} i \xleftrightarrow{\text{相互等价}} * \&i$;
2. $\&*p \xleftrightarrow{\text{相互等价}} p$;
3. $\&*i$ 是非法的表达式, 因为 i 不是地址, 也不是指针变量。

前面叙述了当指针变量 p 指向一个变量 i 时, 可以用 $*p$ 的形式存取该变量 i 的数据。实际上只要指针变量 p 指向内存的某一存储单元, 就可用 $*p$ 的形式存取该存储单元的数据。例如下面的程序段:

```
int *p;
p = (int *) malloc(sizeof(int));
*p = 789;
```

上述程序段第二条语句在内存动态存储区分配一块 2 字节的存储区, 并把其首地址赋值给指针变量 p , 即 p 指向这一存储区, 第三条语句把整数 789 存入这一存储区。

另外, 指针运算符“*”是单目运算, 指针运算符的运算对象必须是指针变量或地址。而乘法运算符“*”是双目运算符, 它的运算对象可以是实型数据或整型数据。

[例 5.1] 由键盘输入一个正整数, 求出其最高位数字。用指针变量来完成本题。

```
main()
{
    long i, *p;
    p = &i;
    printf("请输入一个正整数:");
    scanf("%ld", p); /* 本语句等价于 scanf("%ld", &i); */
    while (*p >= 10) *p /= 10; /* 求出该正整数的最高位数字 */
    printf("最高位数字是:%ld.\n", *p);
}
```

运行结果:

请输入一个正整数:56789

最高位数字是:5。

上例说明当指针变量 p 指向变量 i , 则 $*p$ 就与变量 i 等价。初学指针时应特别注意: 这里赋初值语句 $p = \&i$; 是不可少的, 若指针变量 p 在使用之前未赋初值, 它可能指向内存中任意一个地址, 由键盘输入的数据将代替该地址处的内容, 可能导致程序出现意想不到的错误。

5.2.2 无类型指针

ANSI 新标准增加了一种无类型指针“ $\text{void } *$ ”。可定义一个指针变量, 指向 $\text{void } *$ 类型, 在将它的值赋给另一个指针变量时, 要进行强制类型转换。例如, 有如下程序段:

```
int *p1;
```

```

char * p2;
void * q;          /* 定义 q 为无类型指针变量 */
q = malloc(sizeof(int));
p1 = (int *)q;      /* 将 q 的地址赋值给指针变量 p1, q 与 p1 指向同一地址 */
* p1 = 0x4142;      /* 对 p1 所指的内存赋值 0x4142 (即十进制数 16706) */
p2 = (char *)q;     /* 将 q 的地址赋值给指针变量 p2, q 与 p2 指向同一地址 */
printf("%d, %c, %c\n", * p1, * p2, * (p2+1));
free(q);

```

以上程序段运行的结果是:

16706, B, A

从上面的程序段可知,可以用无类型指针 q 指向内存中的一块存储单元,当用不同类型的指针与 q 指向同一地址时,可存取不同类型的数据。若指向整型类型的指针 $p1$ 指向 q 所指的地址时,则可用 $*p1$ 存取整型数据;若指向字符型的指针 $p2$ 指向 q 所指的地址时,则可用 $*p2$ 存取字符型数据,上述程序段中,0x41 (即十进制数 65) 是字符 'A' 的 ASCII 码。

同样也可以用强制类型转换,把一个指针的值赋值给一个无类型指针。例如:

```

float f, * p3 = &f;
void * q;
q = (void *)p3;

```

则此时 q 和 $p3$ 都指向实型变量 f 。

5.3 指针与一维数组

5.3.1 指针与一维数组

1. 一维数组和数组元素的地址

一个数组的元素在内存中是连续存放的,数组第一个元素的地址称数组的首地址。在 C 语言中,数组名是该数组的首地址。例如有以下定义语句:

```
int a[10], * p;
```

则语句 $p=a$; 和 $p=\&a[0]$; 是等价的,都表示指针变量 p 指向 a 数组的首地址。数组首地址的值在 C 语言中是一个地址常量,是不能改变的。因此,语句 $a=p$; 或 $a++$; 都是非法的。

若数组的首地址是 a ,且指针变量 p 指向该数组的首地址(即 $p=a$;),则 C 语言还规定:

```

数组的第 0 个元素 a[0] 的地址是 a      (等价于 p);
数组的第 1 个元素 a[1] 的地址是 a+1    (等价于 p+1, 详见 5.3.2 的内容);
数组的第 2 个元素 a[2] 的地址是 a+2    (等价于 p+2);
...
数组的第 i 个元素 a[i] 的地址是 a+i    (等价于 p+i);
...

```

例如,有下述程序段:

```
int a[6]={1, 2, 3, 4, 5, 6}, *p=a; /* p 指向整型数组 a 的首地址 */
double d[6]={1.1, 2.2, 3.3, 4.4, 5.5, 6.6}, *q=d;
/* q 指向双精度实型数组 d 的首地址 */
```

假设数组 a 的首地址是 2000, 假设数组 d 的首地址是 3000, 则上述两个数组分配的内存情况如图 5.4 所示, 应注意整型数组(int)每下移一个元素地址加 sizeof(int) 即 2 字节, 双精度实型数组(double)每下移一个元素地址加 sizeof(double) 即 8 字节。

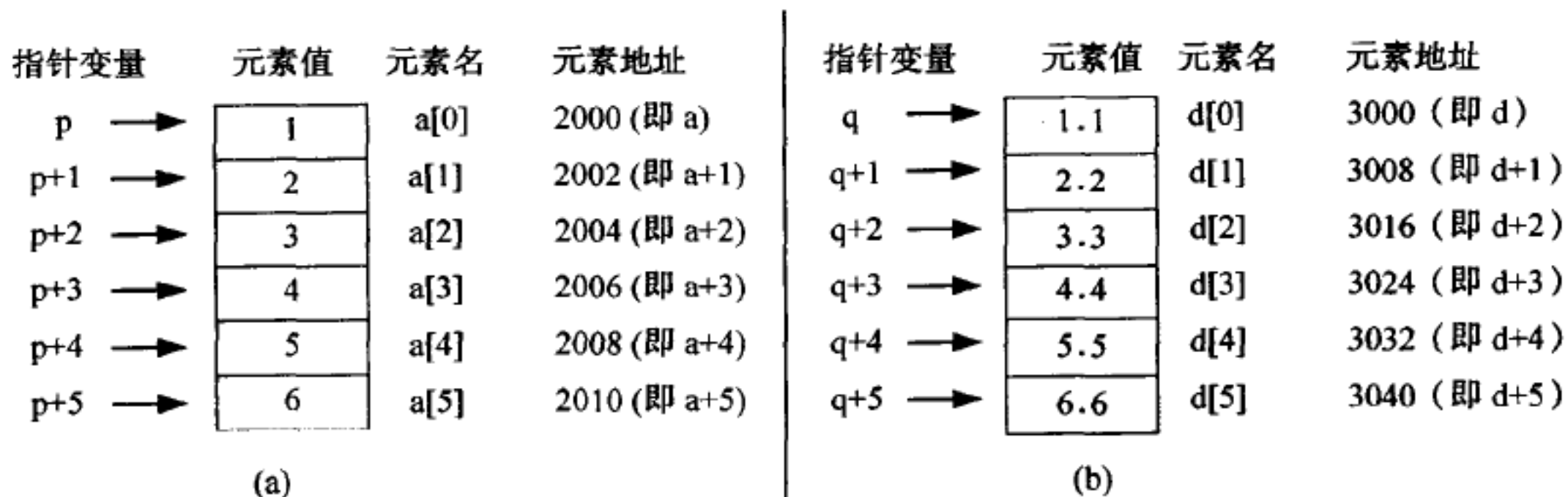


图 5.4 一维数组元素地址的表示形式

2. 通过一维数组名所代表的地址存取数组元素

假设已定义一维数组 a, 由上述可知 a+i 是元素 a[i] 的地址, 根据指针运算符“*”的运算规则知 *(a+i) 与元素 a[i] 等价。例如, 下述程序段:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
*(a+5) = 50; /* 相当于 a[5] = 50; */
scanf("%d", &a[8]); /* 相当于 scanf("%d", a+8); */
printf("%d\n", *(a+5)); /* 相当于 printf("%d\n", a[5]); */
```

3. 通过指针运算符“*”存取数组元素

设有如下程序段:

```
int a[10], *p;
p = a;
```

即 p 指向 a 数组的首地址, 由上述可知 p+i 是元素 a[i] 的地址, 根据指针运算符“*”的运算规则知 *(p+i) 与元素 a[i] 等价。例如, 下述程序段:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, *p = a;
*(p+5) = 50; /* 相当于 a[5] = 50; */
scanf("%d", &a[8]); /* 相当于 scanf("%d", p+8); */
printf("%d\n", *(p+5)); /* 相当于 printf("%d\n", a[5]); */
```

4. 通过带下标的指针变量存取数组元素

C 语言中的下标运算符“[]”可以构成表达式, 假设 p 为指针变量, i 为整型表达式, 则可以把 p[i] 看成是表达式, 首先按 p+i 计算地址, 然后再存取此地址单元中的值。因此 p[i] 与 *(p+i) 等价。例如, 下述程序段:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, *p = a;
p[5] = 50; /* 相当于 a[5] = 50; */
```

```
scanf("%d", &a[8]);           /* 相当于 scanf("%d", &p[8]); */
printf("%d\n", p[5]);         /* 相当于 printf("%d\n", a[5]); */
```

综上所述,若定义了一维数组 a 和指针变量 p ,且 $p=a$,则有如下等价规则:

$a[i]$ $\xleftrightarrow{\text{相互等价}}$ $p[i]$ $\xleftrightarrow{\text{相互等价}}$ $*(a+i)$ $\xleftrightarrow{\text{相互等价}}$ $*(p+i)$

以上假设 p 所指的数据类型与数组 a 元素的数据类型一致, i 为整型表达式。此处,两个表达式“相互等价”仅仅是指“存取变量值”时两个表达式是等价的。

5.3.2 移动指针及两指针相减运算

1. 移动指针

移动指针就是把指针变量加上或减去一个整数,或通过赋值运算使指针变量指向邻近的存储单元。因此,只有当指针变量指向一片连续的存储单元(通常是数组)时,指针的移动才有意义。若有下述程序段:

```
类型名 *p, *q;
p=p+n;
q=q-m;
```

此处“类型名”是指针变量 p 、 q 所指向变量的数据类型,并假定 m 、 n 为正整数,则系统将自动计算出:

p 指针向高地址方向位移的字节数 $= \text{sizeof}(\text{类型名}) * n$;

q 指针向低地址方向位移的字节数 $= \text{sizeof}(\text{类型名}) * m$;

指针变量每增 1、减 1 运算一次所位移的字节数等于其所指的数据类型的大小,而不是简单地把指针变量的值加 1 或减 1,见[例 5.2]。

2. 两指针相减运算

指向同一块连续存储单元(通常是数组)的两个指针变量可以进行相减运算。假设指针变量 p 、 q 指向同一数组,则 $p-q$ 的值等于 p 所指对象与 q 所指对象之间的元素个数,若 $p > q$ 则取正值, $p < q$ 取负值,见[例 5.2]。

指针变量除了可以“位移”和“相减”之外,不能作其他算术运算。

[例 5.2] 本例说明指针变量“位移”和“相减”的操作。

```
main()
{
    int a[6]={1, 2, 3, 4, 5, 6}, *p1=a, *p2;
    /* p1 指向整型数组 a 的首地址 */
    double d[6]={1.1, 2.2, 3.3, 4.4, 5.5, 6.6}, *q1=d, *q2;
    /* q1 指向双精度实型数组 d 的首地址 */
    p1++; /* p1 指向 a 数组的元素 a[1], p1 向下移一个元素(2 个字节) */
    p2=p1+3; /* p2 指向 a 数组的元素 a[4], p2 与 p1 相距 3 个元素 */
    printf("%d\n", p1-p2); /* 输出 -3 到屏幕,因为 p2 与 p1 相距元素的个数为 3 */
    q1++; /* q1 指向 d 数组的元素 d[1], q1 向下移一个元素(8 个字节) */
    q2=q1+3; /* q2 指向 d 数组的元素 d[4], q2 与 q1 相距 3 个元素 */
    printf("%d\n", q2-q1);
```



```

/* 输出 3 到屏幕,因为 q2 与 q1 相距元素的个数为 3 */
printf("%d, %d, %3.1f, %3.1f\n", *p1, *p2, *q1, *q2);
}

```

运行结果:

```

-3
3
2, 5, 2.2, 5.5

```

上例运行后指针指向情况见图 5.5(a) 和图 5.5(b),图中假设数组 a 的首地址是 2000,假设数组 d 的首地址是 3000。

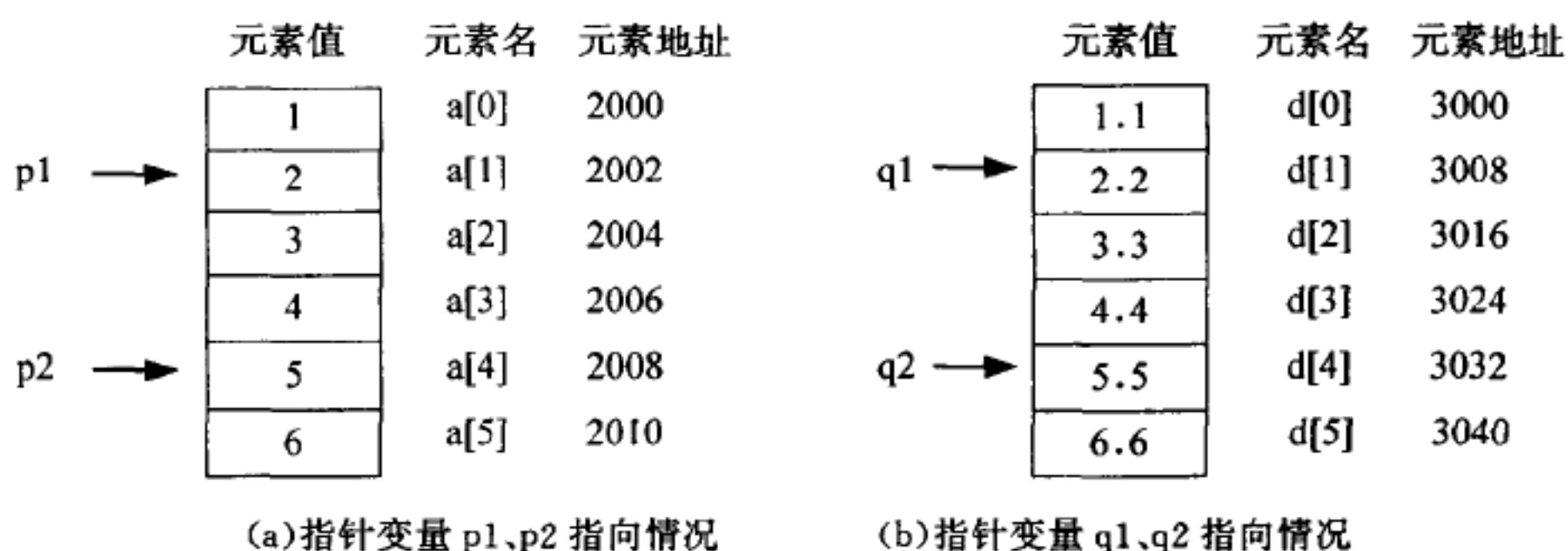


图 5.5 [例 5.2] 中指针变量的指向情况

在[例 5.2]中,指针变量 p1、p2 所指向的数据类型是整型,这时 p1、p2 每位移一个元素,将使指针的地址值位移两 2 字节,设初始时 p1 的值为 2000,则执行语句 p1++;后, p1 的值为 2002,见图 5.5(a);而指针变量 q1、q2 所指向的数据类型是双精度实型,这时 q1、q2 每位移一个元素,将使指针的地址值位移 8 个字节,设初始时 q1 的值为 3000,则执行语句 q1++;和 q2=q1+3;后,q2 的值为 3032,见图 5.5(b)。

5.3.3 指针比较

指向同一块连续存储单元(通常是数组)的两个指针变量可以进行关系运算。假设指针变量 p、q 指向同一数组,则可用关系运算符“<”、“>”、“>=”、“<=”、“==”、“!=”进行关系运算。若 p==q 为真,则表示 p、q 指向数组的同一元素;若 p 指向地址较大元素,q 指向地址较小的元素,则 p>q 的值为 1(真),且 p<q 的值为 0(假)。任何指针变量都可以与空指针 NULL 相比较,如语句 if ((p=(double *)malloc(sizeof(double)))!=NULL) *p=123.456;功能是在动态存储区中分配一个连续 8 个字节大小的存储区,如果分配成功,则把 123.456 存入该存储区。

[例 5.3] 输出数组的全部元素。

```

main()
{
    int i, a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }, *p=a;
    printf("\n");
}

```



```

    for(i=0; i<10; i++) printf("%8d", a[i]);           /* 循环语句 1 */
    for(i=0; i<10; i++) printf("%8d", *(a+i));         /* 循环语句 2 */
    for(i=0; i<10; i++) printf("%8d", *(p+i));         /* 循环语句 3 */
    for(i=0; i<10; i++) printf("%8d", p[i]);           /* 循环语句 4 */
    for(i=0, p=a; i<10; i++) printf("%8d", *p++);      /* 循环语句 5 */
    for (p=a; p<a+10; p++) printf("%8d", *p);         /* 循环语句 6 */
}

```

上例中的六个循环语句功能是一样的,都用来输出 a 数组的 10 个元素。第 5、6 条循环语句使用了 p++ 移动指针,使指针不断地指向下个元素。注意第 6 条循环语句采用了表达式 p<a+10 这样的指针比较运算。由于第 5 条循环语句使指针 p 产生了位移,因此第 6 条循环语句中的赋初值 p=a 表达式不可缺少。

[例 5.4] 指针运算符“*”与增 1 运算符“++”同时作用于一个指针变量的情况。

```

main()
{
    int i, a[]={ 11, 22, 33, 44, 55, 66 }, *p=a;
    printf("%3d,", (*p)++);
    /* 先输出 *p,即输出 a[0],再进行(*p)++,即 a[0]的值变成12,p 值不
       变,仍指向 a[0] */
    printf("%3d,", *p++);
    /* *p++相当于*(p++),先输出 *p,即输出 a[0],再进行 p++,p 指
       向 a[1] */
    printf("%3d,", *++p);
    /* *++p 相当于*(++p),先进行++p,p 指向 a[2],再输出 *p,即输
       出 a[2] */
    printf("%3d\n", ++*p);
    /* ++*p 相当于++(*p),先进行++(*p),即 a[2]的值变成34,再
       输出 *p, p 仍指向 a[2] */
    for (p=a; p<a+6; p++) printf("%3d,", *p);
    /* 最后输出数组的全部元素 */
    printf("\n");
}

```

运行结果:

11, 12, 33, 34

12, 22, 34, 44, 55, 66,

在分析[例5.4]时,应掌握如下规则:

1. 指针运算符“*”与增1运算符“++”均为单目运算符,运算优先级相同,结合方向是“从右到左”;

2. 增1运算符“++”作前缀运算符是“先加1,后使用”,增1运算符“++”作后缀运算符是“先使用,后加1”。

指针运算符“*”与减1运算符“--”同时作用于一个指针变量的情况与上述类似,读者可

以自己举例说明。

5.3.4 字符串

C语言中可以用两种方式来处理字符串:用字符数组来处理字符串以及用指针来处理字符串。

1. 字符数组

4.1.2节中已经讲述了字符数组的概念。每个元素都是字符类型的数组称为字符数组。它的定义形式和元素的存取方法与一般数组相同。

[例5.5] 用字符数组存储字符串与字符。

```
main()
{   char str[]={'O','k','!','\0'}; /* 数组名为str,由四个元素组成,每个元素都存
                                     放相应的字符,如s[3]的值为'\0' */
    char str1[]="Ok!";
    /* 数组名为str1,由四个元素组成,元素赋值情况与上面的str数组相同 */
    char str2[]={'O','k','!'}; /* 数组名为str2,由三个元素组成,其中str2[0]值
                                   为'O',str2[1]值为'k',str2[2]值为'!' */

    printf("%s",str);
    printf("%s\n",str1);
    printf("%c%c%c\n",str2[0],str2[1],str2[2]);
}
```

运行结果:

Ok!Ok!

Ok!

上例在定义字符数组的同时使用了两种赋初值的方式:

(1)将字符逐个赋给数组中各个元素;

(2)直接用字符串常量给字符数组赋初值,系统将字符串中的字符顺序逐个赋给数组中各个元素后,紧接着再把字符串结束标志'\0'赋值给后面的元素。

在[例5.5]中,由于字符数组str2最后未赋字符串结束标志'\0',因此不能视为字符串,不能整体输出。

使用字符数组来表示字符串时,应注意:

(1)字符数组与字符串的区别在于:每个字符串的最后都会加上一个字符'\0',而字符数组的每个元素用于存放一个字符,并没有规定最后一个字符的内容。若在字符数组中存放一系列字符之后,接着加放一个字符'\0',这就等价于在字符数组中存放了一个字符串。如果在字符数组中没有存放字符串结束标志'\0',该字符序列就不是字符串。

(2)若要使用字符数组存放字符串,在定义字符数组时,其元素的个数至少应该比字符串的长度多1,要留有存放字符串结束标志'\0'的元素。

(3)在定义字符数组时,可以用两种方式把一个字符串赋值给字符数组(见[例5.5]):

①将字符串中的字符依次逐个赋给数组中各个元素后,接着加放一个字符'\0';

②直接用字符串常量给字符数组赋初值,系统将字符串中的字符顺序逐个赋给数组中各

个元素后,自动把字符串结束标志'\0'赋值给后面紧接着的元素。

(4)非定义语句中,不能用赋值运算符“=”把字符串赋值给字符数组。例如:

```
char str[10];
str="Ok!";
```

这里 str 是数组名,是地址常量,其值不能改变。因此,上述第二条赋值语句 str="Ok!"; 是非法的。程序运行时,若要把一个字符串赋值给字符数组,通常用以下两种方式:

①给字符数组元素逐个赋值,最后加放一个字符串结束标志,下述程序段把字符串"Ok!"赋值给字符数组 s。

```
char s[10];
s[0]='O'; s[1]='k'; s[2]='!'; s[3]='\0';
```

用这种方法把字符串赋值给字符数组,很繁琐也不直观。更常用的方法是使用函数把字符串赋值给字符数组。

②使用 C 系统提供的库函数把字符串赋值给字符数组,下述程序段把字符串"Ok!"赋值给字符数组 s。

```
char s[10];
strcpy(s, "Ok!");
```

类似的函数还有 strncpy、strcat 等(参见第六章与附录 F)。也可使用输入函数 scanf、gets 把从键盘输入的字符串赋值给字符数组。

(5)允许使用输入、输出库函数(如 gets、puts、scanf、printf)对字符数组中的字符串整体输入、输出。

(6)对于较长的字符串,Turbo C 允许使用下述分行方法来书写长字符串:

```
char str2[5000];
char str1[]="    科学是关于自然的知识总体,它代表了人类的共同努力、洞察"
            "力、研究成果和智慧。\n 当人们最初发现了在他们周围反复出现的"
            "各种关系时,就有了科学的开端。\n";
strcpy(str2, "通过对这些关系的仔细观察,人们开始了解了自然,"
            "而由于自然的可靠性,\n 人们还发现他们能够作出预测,"
            "从而有可能在某种程度上控制他们周围的环境。\n");
```

2. 字符指针

字符串常量是由双引号括起来的字符序列,例如"Ok!"。程序中如果出现字符串常量,系统就在内存中给该字符串常量分配一连续的存储区,用以存放该字符串,系统还自动在该字符串的末尾加上字符串的结束标志'\0',因此一个字符串常量所占存储区的字节数比它的长度(字符的个数)多一个字节。可以用字符指针指向字符串,然后通过字符指针来存取字符串。例如:

```
char * sp;
sp="Ok!";
```

C 语言中把字符串常量赋值给字符指针变量,相当于把该字符串常量的首地址赋值给字符指针变量,上述语句 sp="Ok!"; 使 sp 指向字符串常量"Ok!",如图 5.6 所示(图中假设字符串常量"Ok!"的首地址是 1500)。

当 sp 指向某字符串常量时,就可以用 sp 来存取该字符串,sp[i]或 *(sp+i)就相当于字

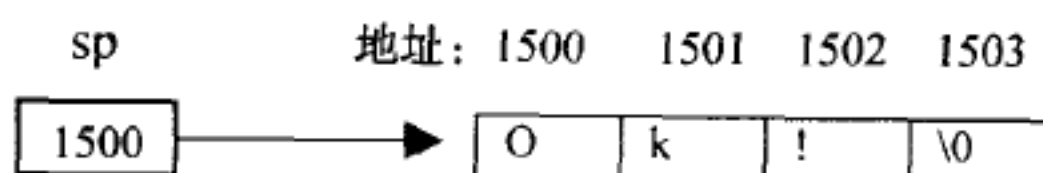


图5.6 字符串常量"Ok!"存放于内存中,字符指针 sp 指向字符串常量"Ok!"

字符串的第 $i+1$ 个字符,如上述 $sp[2]$ 的值为 '!'。

[例5.6] 利用字符指针把字符串 s1 复制到字符串 s2。

```
main()
{ char s1[]="Good!", s2[15]="How are you!", *from=s1, *to=s2;
  while (*from) *to++=*from++;
  *to='\0';
  printf("%s\n%s\n",s1,s2);
}
```

运行结果:

Good!

Good!

图5.7说明了上例中复制前、后字符数组 s2 的变化情况,复制后输出 s2 时,遇第一个字符串结束标志($s2[5]$ 的值为 '\0')即停止输出。元素 $s[13]$ 与 $s[14]$ 未用到,其值是任意字符。

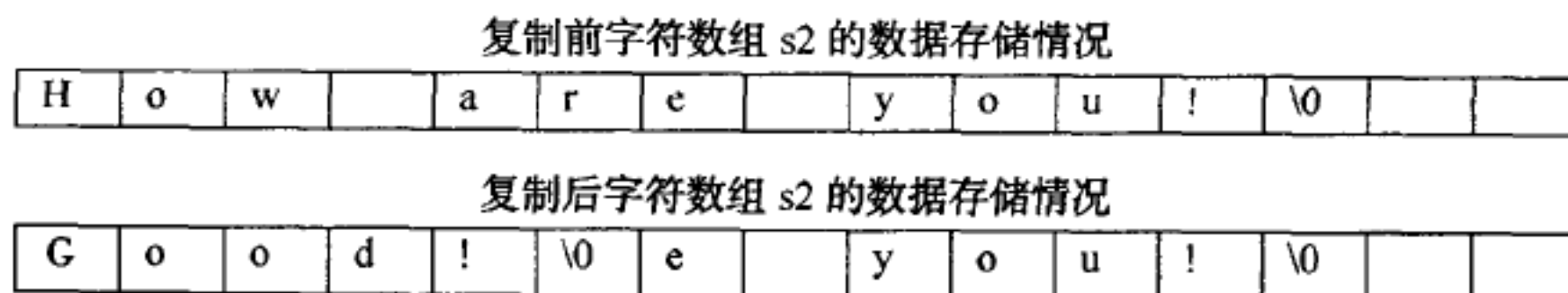


图5.7 [例5.6]中字符数组 s2 复制前后的变化情况

3. 字符数组与字符指针的区别

表5.1总结了使用字符数组处理字符串与使用字符指针处理字符串的区别。

表5.1 使用字符指针处理字符串与使用字符数组处理字符串的区别

使用字符指针处理字符串	使用字符数组处理字符串
值存放的是字符串首地址	由各元素组成,每个元素放一个字符。
<code>char *sp="Ok!";</code> 语句在内存中为字符串常量"Ok!"分配四个字节的一连续的存储区存储该字符串,在末尾加 '\0',并把字符串的首地址赋值给字符指针变量 sp。	<code>char s[]="Ok!";</code> 语句定义一个数组名为 s,由四个元素组成的数组,每个元素都存放相应的字符,如 $s[0]$ 的值为 'O', $s[1]$ 的值为 'k', $s[2]$ 的值为 '!', $s[3]$ 的值为 '\0'。
<code>char *sp; sp="Ok!";</code> 这两条语句的作用与 <code>char *sp="Ok!";</code> 等价。	<code>char *s[50]; s="Ok!";</code> 此处第二条语句是非法的,数组名 s 是数组的首地址,是常量,不能被赋值。

(续表)

使用字符指针处理字符串	使用字符数组处理字符串
char *sp; scanf("%s", sp); 此处字符指针变量 sp 未赋初值,它可能指向内存中任意一个地址,由键盘输入的字符串将代替该地址处的内容,可能导致意想不到的错误。正确的使用方式是先让 sp 指向一字符数组: char str[300], *sp = str; scanf("%s", sp);。类似这样的输入函数还有 gets、strcpy、strcat 等,都要注意字符指针变量赋初值问题。	char *s[300]; scanf("%s", s); 此处的 scanf 函数的用法是正确的,由键盘输入的字符串将逐字存入字符数组 s 中,系统自动在输入的字符串最后加'\0'表示字符串结束。
char *sp; sp 是字符指针变量,其值可以改变。例如 sp="Good morning!"; sp=sp+5; printf("%s\n", sp); 将输出 morning!	char s[300]; s 是数组名,是数组的首地址,是地址常量,其值不可改变。

[例5.7] 删除一个字符串中所有的空格字符。

```
#include "stdio.h"
main()
{
    char s[500], *p1, *p2;
    printf("请输入一个字符串,该字符串内的空格将被删除:");
    gets(s);
    p1=p2=s;
    while( *p1 )
        if( *p1 == ' ' ) p1++;
        else *p2++ = *p1++;
    *p2='\0';
    printf("删除空格后的字符串是:%s\n", s);
}
```

运行结果:

请输入一个字符串,该字符串内的空格将被删除:How are you!

删除空格后的字符串是:Howareyou!

分析上例的算法:首先 p1、p2 指向字符数组 s 的首地址,然后用字符指针 p1 对字符数组 s 中的字符依次逐个检查,如果是空格就继续检查下一个字符;如果不是空格就把字符存放在原 s 数组中,所放的位置由字符指针 p2 决定,然后 p1、p2 下移一个元素;最后再添加上字符串结束标志,如图 5.8 所示。其中语句 *p2++ = *p1++; 与复合语句 { *p2 = *p1; p2++; p1++; } 等价。

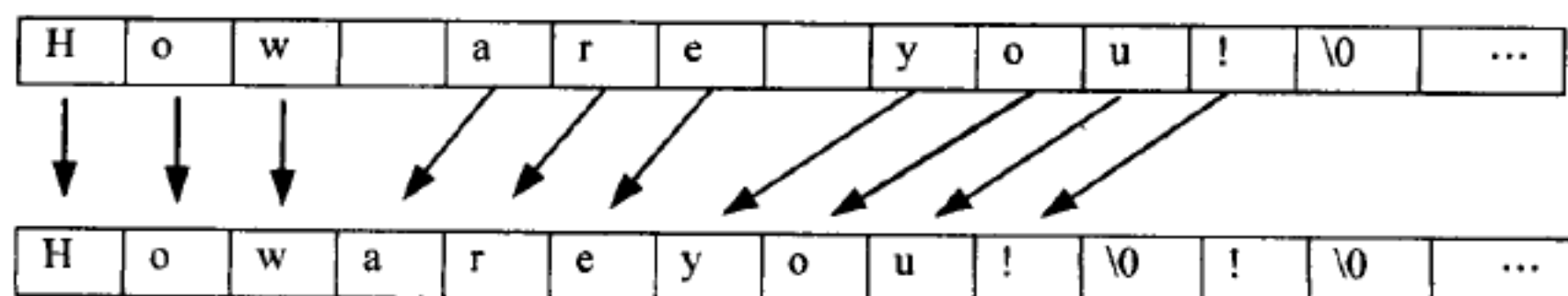


图5.8 在一个字符数组中删除所有空格字符

5.3.5 指针与二维数组

1. 二维数组和数组元素的地址

C 语言的二维数组由若干个一维数组构成。即先把二维数组视为一个一维数组,而这个一维数组的每一个元素又是一个一维数组。例如定义以下二维数组:

```
int a[3][5] = { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9, 10 }, { 11, 12, 13, 14, 15 } };
```

a 为二维数组, a 数组有3行5列,共15个元素。可以这样理解:数组 a 由三个元素组成: $a[0]$ 、 $a[1]$ 、 $a[2]$,而每个元素又是一个一维数组,且都含有5个元素,即:

$a[0]$ 的元素有 $a[0][0]$ 、 $a[0][1]$ 、 $a[0][2]$ 、 $a[0][3]$ 、 $a[0][4]$,数组名 $a[0]$ 是这一数组的首地址;

$a[1]$ 的元素有 $a[1][0]$ 、 $a[1][1]$ 、 $a[1][2]$ 、 $a[1][3]$ 、 $a[1][4]$,数组名 $a[1]$ 是这一数组的首地址;

$a[2]$ 的元素有 $a[2][0]$ 、 $a[2][1]$ 、 $a[2][2]$ 、 $a[2][3]$ 、 $a[2][4]$,数组名 $a[2]$ 是这一数组的首地址。

同样,数组名 a 就是数组 $\{a[0]$ 、 $a[1]$ 、 $a[2]\}$ 的首地址,如图5.9和图5.10所示。

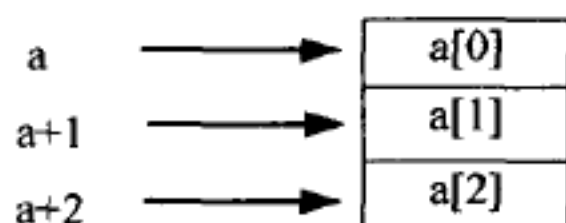
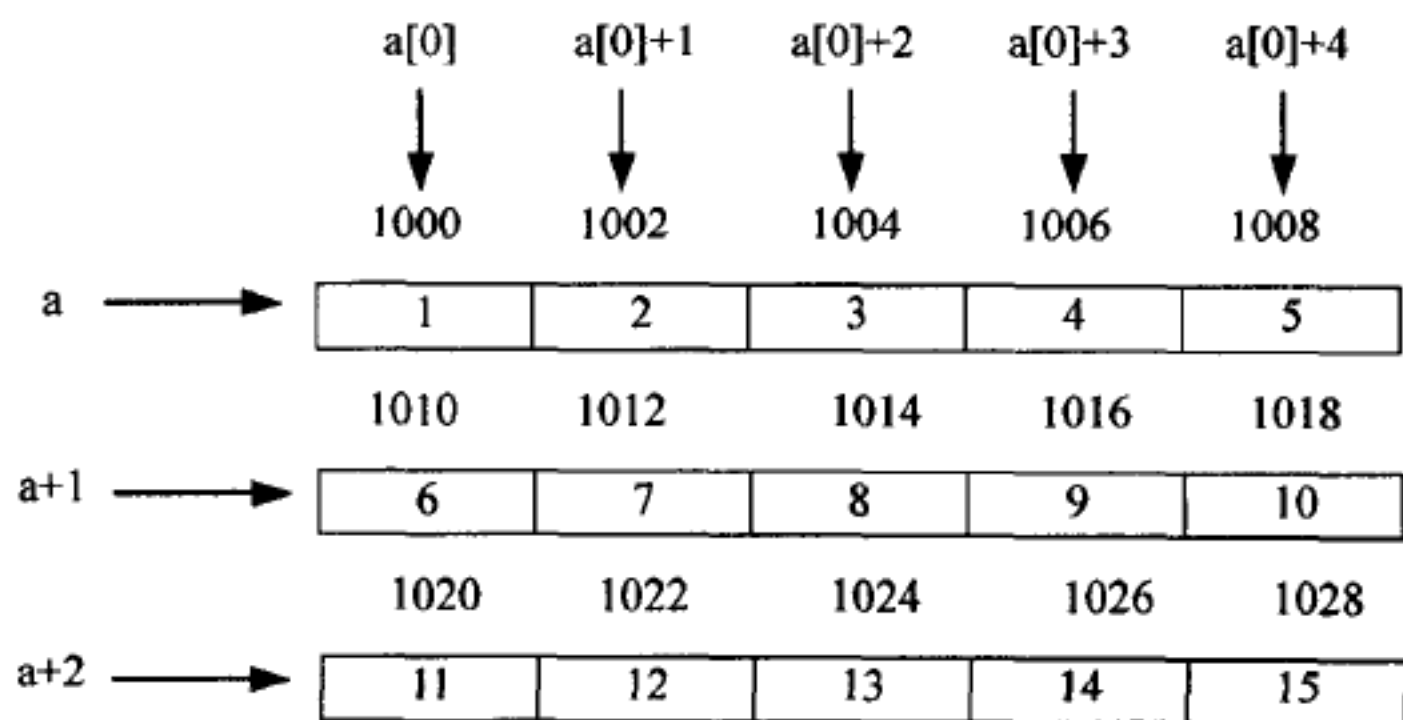


图5.9 多个一维数组构成二维数组

图5.10 二维数组元素的地址(假设 $a[0][0]$ 的地址是1000)

在 C 语言中,二维数组名 a 同样是二维数组的首地址,其值为二维数组中第一个元素的地址。 a 也是二维数组第0行的首地址, $a+1$ 是第1行的首地址, $a+2$ 是第2行的首地址。假设此二维数组的首地址为1000,由于每行有5个整型元素,所以 $a+1$ 值为1010, $a+2$ 值为1020,如图5.10所示。

由于 $a[0]$ 、 $a[1]$ 、 $a[2]$ 是一维数组名,它们就是对应数组的首地址。因此:

$a[0]$ 是第0行第0列元素的地址 ($\&a[0][0]$), $a[0]+1$ 是第0行第1列元素的地址

$(\&a[0][1])\dots\dots$

$a[1]$ 是第 1 行第 0 列元素的地址 ($\&a[1][0]$), $a[1]+1$ 是第 1 行第 1 列元素的地址 ($\&a[1][1]$) $\dots\dots$

$a[i]+j$ 是第 i 行第 j 列元素的地址 ($\&a[i][j]$)。

另外, 由指针运算符“ $*$ ”和下标运算符“ $[]$ ”的运算规则得知: $a[0]$ 与 $*(a+0)$ 等价, $a[1]$ 与 $*(a+1)$ 等价 $\dots\dots a[i]$ 与 $*(a+i)$ 等价, $a[i]+j$ 与 $*(a+i)+j$ 等价, 它们的值都是 $\&a[i][j]$, 即元素 $a[i][j]$ 的地址。

表 5.2 总结了二维数组元素的地址表示形式, 表中假设二维数组 a 每行有 5 个元素, 并假设 a 的首地址为 1000。

表 5.2 二维数组元素的地址表示形式(此处假设二维数组 a 每行有 5 个元素)

表示形式	含 义	地 址
a	二维数组名, 数组首地址	1000
$a+i$	第 i 行的首地址	$1000+5*i*\text{sizeof(类型)}$
$a[i]+j, *(a+i)+j, \&a[i][j]$	第 i 行第 j 列元素的地址	$1000+(5*i+j)*\text{sizeof(类型)}$
$\&a[0][0]+5*i+j$	第 i 行第 j 列元素的地址, 此处假设 a 数组每行有 5 个元素	$1000+(5*i+j)*\text{sizeof(类型)}$

执行语句 `printf("%p,%p", a, *a);`, 可知 a 和 $*a$ 值是相同的, 它们都指向同一地址, 但 a 是行指针, $a+1$ 指向下一行; 而 $*a$ 即为 $a[0]$, 是数组 a 第 0 行第 0 列元素的地址 $\&a[0][0]$, $*a+1$ 指向下一个元素 $a[0][1]$ 。

2. 通过地址存取二维数组元素

假设有如下定义:

```
int a[3][5], i, j;
```

则二维数组 a 中的任一元素 $a[i][j]$, 可以用下述表达式之一来等价表示:

(1) $*(a[i]+j)$ 由上述知 $a[i]+j$ 是第 i 行第 j 列元素的地址, 因此 $*(a[i]+j)$ 与 $a[i][j]$ 等价;

(2) $*(*(a+i)+j)$ $*(a+i)+j$ 也是第 i 行第 j 列元素的地址, 因此 $*(*(a+i)+j)$ 与 $a[i][j]$ 等价;

(3) $((*(a+i))[j])$ 此表达式表示先取到 $*(a+i)+j$ 处的地址, 再到该地址处存取数据, 因此 $((*(a+i))[j])$ 与 $a[i][j]$ 等价;

(4) $*(\&a[0][0]+5*i+j)$ 由于每行 5 个元素, $\&a[0][0]+5*i+j$ 就是第 i 行第 j 列元素的地址, 因此 $*(\&a[0][0]+5*i+j)$ 也与 $a[i][j]$ 等价。由表 5.2 知 $\&a[0][0]$ 与 $*a$ 或 $a[0]$ 等价, 因此有如下等价关系:

$$\begin{array}{ccccc} *(\&a[0][0]+5*i+j) & \xleftrightarrow{\text{相互等价}} & *(a[0]+5*i+j) & & \\ \xleftrightarrow{\text{相互等价}} & *(&a+5*i+j) & \xleftrightarrow{\text{相互等价}} & a[i][j] \end{array}$$

[例 5.8] 以下通过一个简单的例子来说明上述等价关系。

```
main()
```

```
{
```

```
    int i, j, a[][5]={ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
```

```

printf( "%5d,", *(a[1]+4) );          /* 输出元素 a[1][4] */
printf( "%5d\n", --*( *a+3) );
/* 先计算--a[0][3],再输出 a[0][3]的值 */
(* (a+1))[1] += 5;                    /* 等价于语句 a[1][1] += 5; */
*( &a[0][0]+5*2+3) = 55;              /* 等价于语句 a[2][3] = 55; */
for(i=0; i<3; i++)
{
    for (j=0; j<5; j++) printf( "%5d", a[i][j] );
    printf( "\n" );
}
}

```

运行结果:

```

10,    3
1      2      3      3      5
6      12     8      9      10
11     12     13     55     15

```

上例程序中应用了上述等价表达式来存取二维数组元素。

3. 通过指向数组元素的指针变量存取二维数组元素

假设有如下程序段:

```

int a[3][5], i, j, *p;
p=&a[0][0];

```

则二维数组 a 中的任一元素 $a[i][j]$ 与表达式 $*(p+i*5+j)$ 等价,此时 $a[i][j]$ 也与表达式 $p[i*5+j]$ 等价。这是因为 p 是一个指向整型变量的指针,它也可以指向整型数组元素。 $p+1$ 就指向下一个元素,由于每行5个元素, $p+i*5+j$ 就是第 i 行第 j 列元素的地址。由表5.2知, $a[0]$ 与 $*a$ 都是指向第0行第0列元素的地址(即 $\&a[0][0]$),因此在这里,有如下的语句等价关系。

$p=\&a[0][0];$ $\xleftrightarrow{\text{相互等价}}$ $p=a[0];$ $\xleftrightarrow{\text{相互等价}}$ $p=*a;$

[例5.9] 以下通过一个简单的例子来说明上述等价关系。

```

main()
{
    int i, j, *p, a[][5]={ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
    p = *a;
    printf( "%5d,", *(p+2*5) ); /* 输出元素 a[2][0] */
    printf( "%5d\n", (*(p+8))+1 );
    /* 先输出 a[1][3]的值,再计算 a[1][3]++ */
    *(p+14) += *(p+2*5+3); /* 等价于语句 a[2][4] += a[2][3]; */
    for(i=0; i<3; i++)
    {
        for (j=0; j<5; j++) printf( "%5d", a[i][j] );
        printf( "\n" );
    }
}

```

```

    }
}

```

运行结果:

```

    11,    9
    1      2      3      4      5
    6      7      8     10     10
    11     12     13     14     29

```

上例程序中说明了当指针变量 p 指向二维数组的第一个元素时,表达式 $*(p+i*5+j)$ 同样可以存取二维数组元素 $a[i][j]$ 。

4. 通过指向一维数组的指针变量存取二维数组元素

假设有如下程序段:

```

int a[3][5], i, j, (*pi)[5];
pi=a;

```

这里定义的 $(*pi)[5]$ 中,圆括号 $()$ 内的 $*$ 先与 pi 相结合,说明 pi 是一个指针变量,然后 $(*pi)$ 再与右边的 $[5]$ 相结合,说明指针变量 pi 指向“包含5个整型元素的一维数组”。由图5.9知数组名 a 与 pi 一样,指向“包含5个整型元素的一维数组”。因此,当 $pi=a;$ 时, $pi+1$ 将指向下一行(见图5.11),称 pi 为行指针,也称 pi 是(指向一维)数组(的)指针。

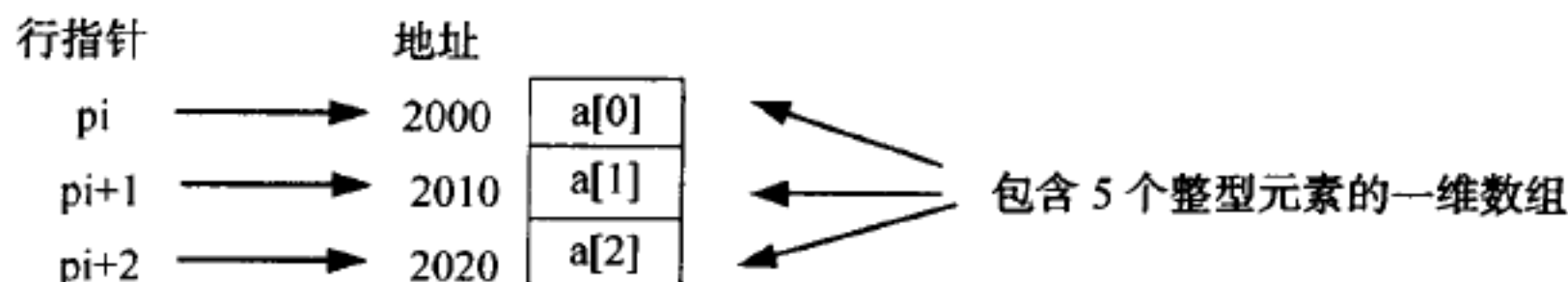


图5.11 指向一维数组的指针变量 pi

按照地址对应关系,二维数组 a 中的任一元素 $a[i][j]$,可以用下述表达式之一来等价表示:

```

(1) * (pi[i]+j) ← 相互等价 → * (a[i]+j) ← 相互等价 → a[i][j]
(2) * (* (pi+i)+j) ← 相互等价 → * (* (a+i)+j) ← 相互等价 → a[i][j]
(3) (* (pi+i))[j] ← 相互等价 → (* (a+i))[j] ← 相互等价 → a[i][j]
(4) pi[i][j] ← 相互等价 → a[i][j]

```

应该注意,这里的 pi 是一个指针变量,其值是可改变的;而 a 是二维数组名,是常量。

[例5.10] 输出一个二维数组的指定行和指定列。

```

main()
{
    int a[][5]={ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
    int i, row, col, (*p)[5];
    p=a;
    printf("请输入二维数组的行(0~2)、列(0~4):");
    scanf("%d%d", &row, &col);
    printf("第%d行的数组元素是:", row);
    for (i=0; i<5; i++) printf("%5d", * (* (p+row)+i) );
}

```

```

printf("\n 第%d 列的数组元素是:", col);
for (i=0; i<3; i++) printf("%5d", *(p[i]+col));
printf("\n");
}

```

运行结果:

请输入二维数组的行(0~2)、列(0~4): 0 3

第0行的数组元素是: 1 2 3 4 5

第3列的数组元素是: 4 9 14

5. 通过指针数组存取二维数组元素

假设有如下程序段:

```

int a[3][5], i, j, *pa[3];
for (i=0; i<3; i++) pa[i]=a[i];

```

这里定义的 `*pa[3]` 中, 下标运算符“`[]`”优先级高于“`*`”运算符, `pa` 先与“`[]`”相结合构成 `pa[3]`, 说明 `pa` 是一个数组名, 它有3个元素, 左边的“`*`”表示 `pa` 的每个元素都是指针, 都可以指向整型变量。for 循环语句把二维数组 `a` 的每行第0列元素的首地址 `a[i]` 赋值给指针变量 `pa[i]`, 如图5.12所示(图中假设数组 `a` 的首地址为2000)。

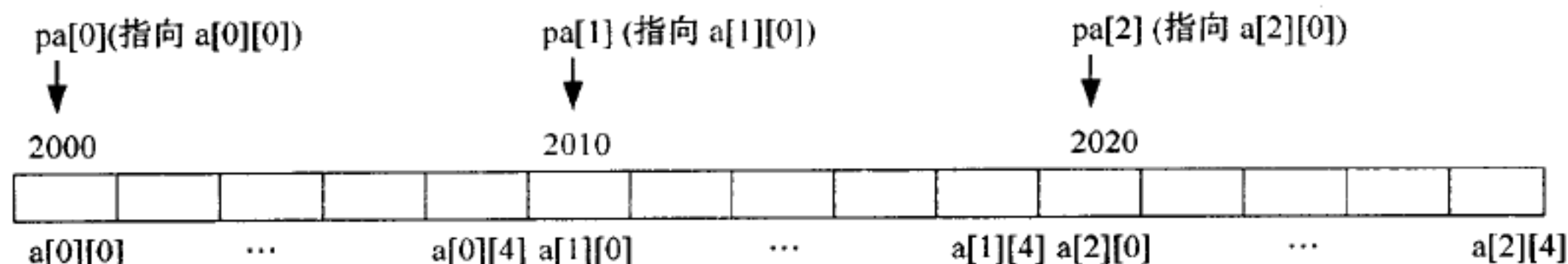


图5.12 用指针数组 `pa` 的各个元素分别指向二维数组 `a` 的每行第0列元素

元素 `pa[i]` (其中 $i=0,1,2$) 是指向整型变量的指针, 可以用它指向整型数组元素。 `pa[i]+1` 就指向下一个元素, 按照地址对应关系, 二维数组 `a` 中的任一元素 `a[i][j]`, 可以用下述表达式之一来等价表示:

$$\begin{aligned}
 (1) & \quad * (pa[i] + j) \xleftrightarrow{\text{相互等价}} * (a[i] + j) \xleftrightarrow{\text{相互等价}} a[i][j] \\
 (2) & \quad * (* (pa + i) + j) \xleftrightarrow{\text{相互等价}} * (* (a + i) + j) \xleftrightarrow{\text{相互等价}} a[i][j] \\
 (3) & \quad (* (pa + i))[j] \xleftrightarrow{\text{相互等价}} (* (a + i))[j] \xleftrightarrow{\text{相互等价}} a[i][j] \\
 (4) & \quad pa[i][j] \xleftrightarrow{\text{相互等价}} a[i][j]
 \end{aligned}$$

应该注意, 上述表达式等价的条件是: 指针数组 `pa` 的每个元素 `pa[i]` 分别指向二维数组 `a` 的第 i 行第0列元素的首地址 `a[i]` (即 `pa[i]=a[i]`, 其中 $i=0,1,2$)。另外, `pa[i]` 是一个指针变量, 其值是可改变的; 而 `a[i]` 是常量, 值不能改变。

[例5.11] 某班有4个学生, 每个学生有5门课程。编程完成下列操作: 1. 求出第3门课程的平均分; 2. 按总分由大到小排序, 输出每位学生的学号、成绩及总分。

```

main()
{
    int i, j;
    float *pa[4], *temp, average=0;

```



```

float score[][7]={ { 1, 76.5, 89.5, 78, 90.5, 66, 0.0 },
                   { 2, 56.5, 69.5, 49, 70.5, 73, 0.0 },
                   { 3, 82, 90, 87.5, 90.5, 81.5, 0.0 },
                   { 4, 67, 69.5, 43, 70.5, 52.5, 0.0 }
                   };

/* 二维数组 socre 用于存放学生的学号(第0列元素)、5门课程的 */
/* 学习成绩(第1列至第5列元素)、每位学生5门课程的总成绩(第6列元素) */
for(i=0;i<4;i++)pa[i]=score[i];/* 给指针数组的每一个元素赋初值 */
for(i=0;i<4;i++)average+=*(pa[i]+3);/* 计算第3门课的总分 */
average/=4.0; /* 计算第3门课的平均分 */
printf("第3门课的平均分是: %6.2f\n", average);
for( i=0; i<4; i++)
    for( j=1; j<6; j++)
        pa[i][6] += ( (*(pa+i))[j] ); /* 计算每位学生5门课的总成绩 */
for( i=0; i<4; i++) /* 采用冒泡法排序 */
    for( j=0; j<=4-i; j++)
        if ( pa[j][6] < pa[j+1][6] ) /* 比较总分值 */
        {
            temp = pa[j];
            pa[j] = pa[j+1];
            pa[j+1] = temp;
        }
/* 以下程序段按总分由大到小顺序, 输出每位学生的学号、成绩及总分 */
for( i=0; i<4; i++)
{
    printf("第%.0f 号学生的成绩为:", pa[i][0]);
    for( j=1; j<6; j++)
        printf( "%-8.2f", *( *(pa+i)+j) );
    printf("总分为: %6.2f\n", pa[i][6]);
}
}

```

运行结果:

第3门课的平均分是: 64.38

第3号学生的成绩为: 82.00 90.00 87.50 90.50 81.50 总分为: 431.50

第1号学生的成绩为: 76.50 89.50 78.00 90.50 66.00 总分为: 400.50

第2号学生的成绩为: 56.50 69.50 49.00 70.50 73.00 总分为: 318.50

第4号学生的成绩为: 67.00 69.50 43.00 70.50 52.50 总分为: 302.50

上例程序说明了当 $pa[i]=score[i]$ (其中 $i=0,1,2,3$) 时, 表达式 $*(pa[i]+j)$ 、 $*(*(pa+i)+j)$ 、 $*(*(pa+i))[j]$ 、 $pa[i][j]$ 及 $score[i][j]$ 均可相互等价, 在排序前可以用 $score[i][j]$ 来代替对应的表达式。指针的优点在于它的值可以改变, 指针可以指向不同的对象, 本例题只需交换指针的值就可完成排序, 排序后将使得元素 $pa[i]$ 的下标 i 值越小, 则 $pa[i]$ 指向总分值越大的那行的首列元素(见图 5.13), 即 $pa[0]$ 指向总分最大的学生的学号, $pa[1]$ 指向总分第二大

的学生的学号……`pa[3]`指向总分最小的学生的学号。本例题如果不用指针,只用二维数组来完成排序,每次交换的数据量是一行的内容,程序的执行效率将大大降低。

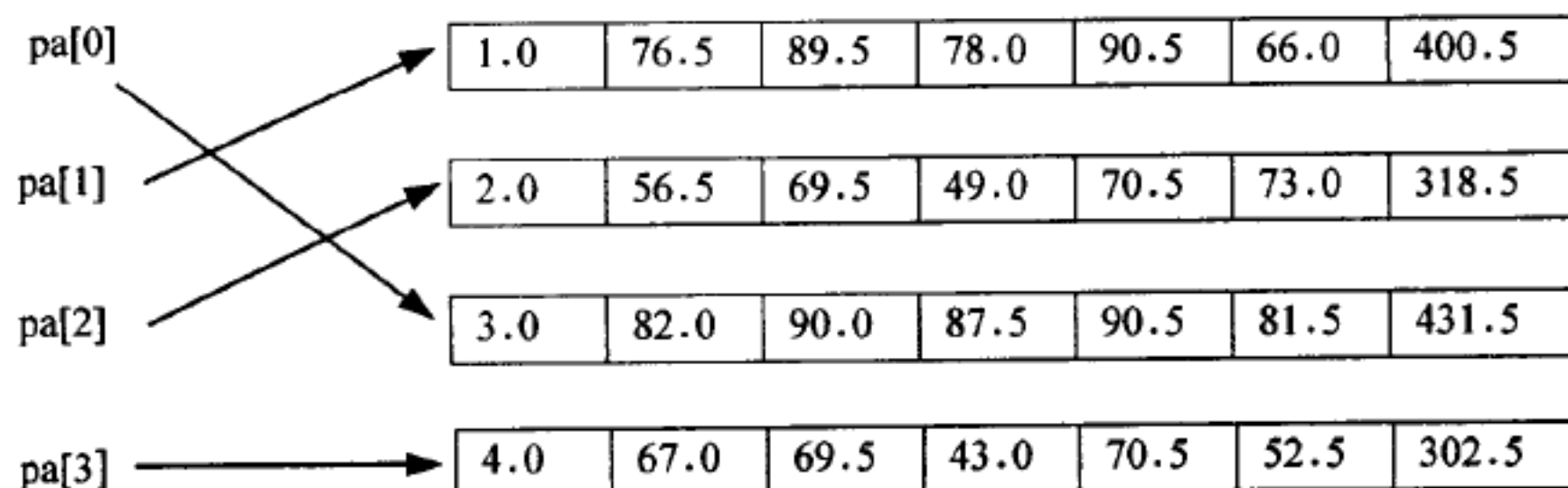


图5.13 排序后指针数组 `pa` 的元素指向二维数组 `score` 各行第0列的情况

6. 通过二维字符数组构成字符串数组

如上所述,C语言中的二维数组可视为一个一维数组,这个一维数组的每个元素又是一个一维数组。因此可以用二维字符数组来构成字符串数组。例如:

```
char book[50][128];
```

数组 `book` 共有50个元素,每个元素可存放128个字符,若用于存放字符串,可以存放50个长度小于128的字符串。字符串数组可以在定义时赋初值。例如下面的定义语句:

```
char str[5][10]={ "BASIC", "ADA", "Pascal", "C", "Fortran" };
```

上述语句的中二维数组的第一维的长度可以省略,系统将根据字符串的个数来分配存储空间,与下面的定义语句等价:

```
char str[ ][10]={ "BASIC", "ADA", "Pascal", "C", "Fortran" };
```

此二维数组 `str` 在内存中存储情况如图5.14所示。`str[i]`(其中 $i=0, 1, 2, 3, 4$)代表了第 i 个字符串的首地址,也可用 `str[i][j]`来存取数组中的一个字符,设有输出语句:

```
printf("%s, %c\n", str[2], str[4][1]);
```

该语句执行后输出结果是:

Pascal, o

即 `str[2]`指向字符串"Pascal",`str[4][1]`的值是'o'。

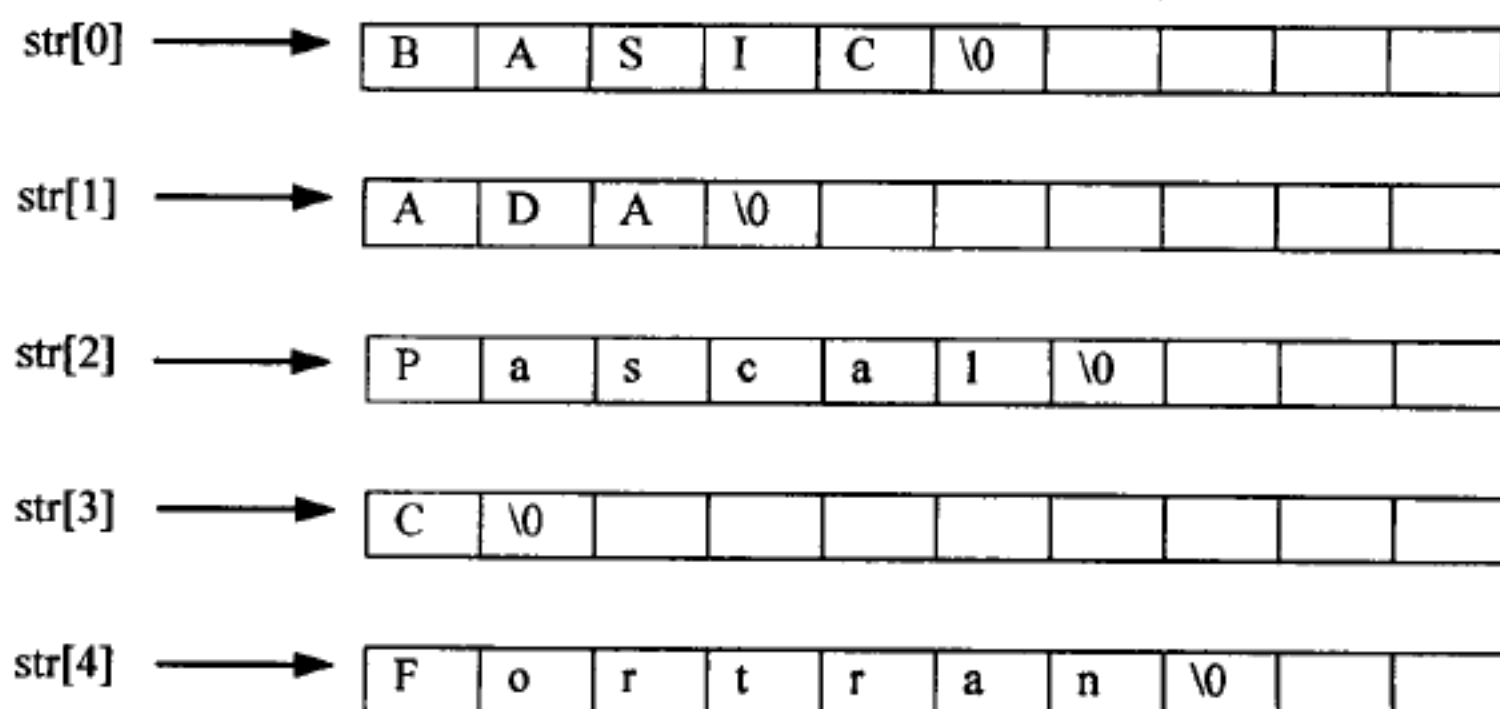


图5.14 二维字符数组构成字符串数组

由图5.14可知,用二维字符数组来存放字符串,各字符串都是从每行的第0个元素开始存放字符,有部分存储单元空闲着,各字符串的长度差别越大,空闲单元就越多,显然这将浪费内存。为了使各字符串常量在内存中存放时不浪费内存空间,可以定义一个指针数组,并在定义时用字符串赋初值的方法,来构成一个字符串数组。例如:

```
char *ps[5]={ "BASIC", "ADA", "Pascal", "C", "Fortran" };
```

这一字符数组在内存的存储情况见图5.15。执行这条语句时,首先为每个字符串常量分配相应的存储空间,然后把相应字符串的首地址赋值给指针数组 ps 的5个对应的元素,即 ps[i](其中 i=0, 1, 2, 3, 4)指向第 i 个字符串,还可以用 p[i][j]、*(p[i]+j)、*(* (p+i)+j)、(* (p+i))[j]等形式存取第 i 个字符串中的第 j 个字符。

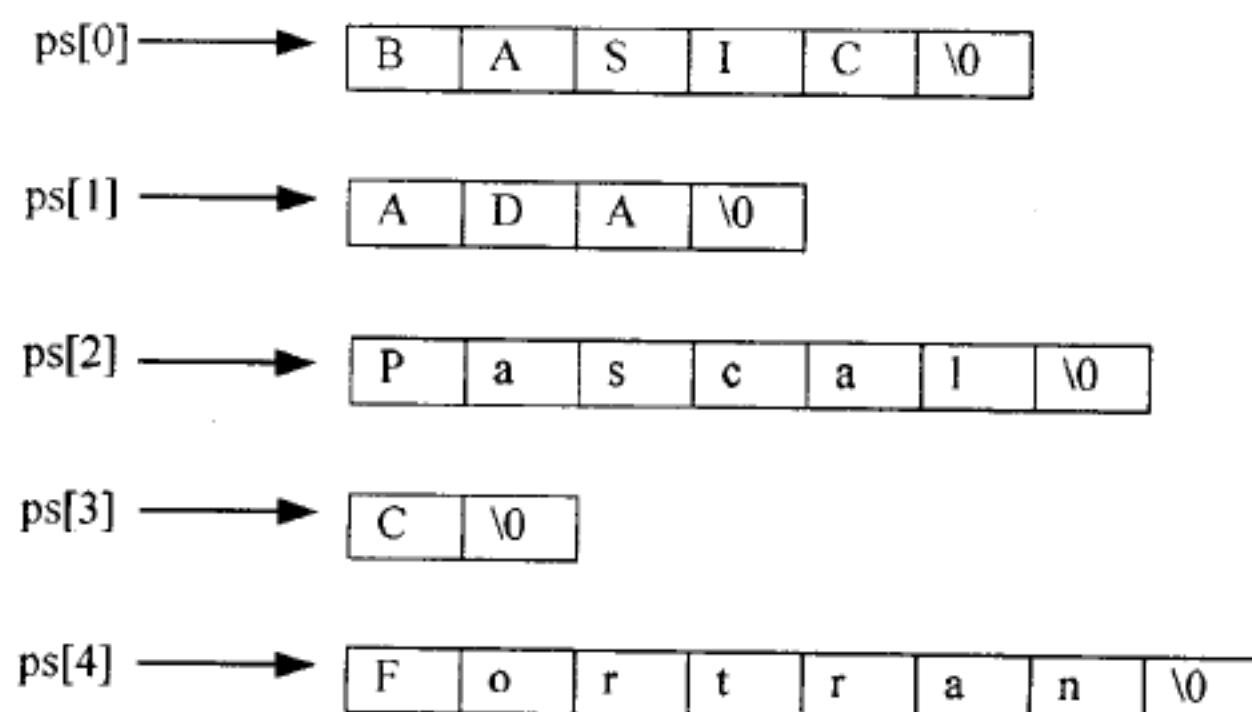


图5.15 指针数组指向字符串数组

设有输出语句

```
printf("%s, %c\n", ps[4], ps[2][1]);
```

该语句执行后输出结果是:

Fortran, a

即 ps[4]指向字符串"Fortran", ps[2][1]的值是'a'。用这种方式组成的字符串数组,系统根据字符串常量的长度来分配存储空间,不会浪费内存空间。

[例5.12] 编写一程序,输入数字星期几,则输出英文对应的星期几。例如,输入"0",则输出"Sunday",若输入"6",则输出"Saturday"。

```

main()
{
    char weeks[][10]={"Sunday", "Monday", "Tuesday", "Wednesday",
                      "Thursday", "Friday", "Saturday"};

    int i;
    do
    {
        printf("请输入星期几(数字0~6):");
        scanf("%d", &i);
    }while(i<0 || i>6);
    printf("%s\n", weeks[i]);
}
  
```

```
}
```

运行结果:

请输入星期几(数字0~6):6

Saturday

上例程序中使用二维数组 weeks 来构成一个字符串数组。程序中还使用了一个 do/while 循环来限制用户输入数字在0~6之间。

[例5.13] 将字符串"BASIC", "ADA", "Pascal", "C", "Fortran"按从小到大的顺序排序后输出。

```
#include "string.h"
main()
{
    char * temp, * ps[5]={"BASIC", "ADA", "Pascal", "C", "Fortran"};
    int i, j, k;
    for(i=0; i<4; i++)
    {
        k=i;
        for (j=i+1; j<5; j++)
            if ( strcmp(ps[k], ps[j])>0) k=j; /* ps[k]总指向值较小的字符串 */
        if ( k!=i )
        {
            temp=ps[i];
            ps[i]=ps[k];
            ps[k]=temp;
        }
    }
    for(i=0; i<4; i++) printf("%s, ", ps[i]);
    printf("%s\n", ps[4]);
}
```

运行结果:

ADA, BASIC, C, Fortran, Pascal

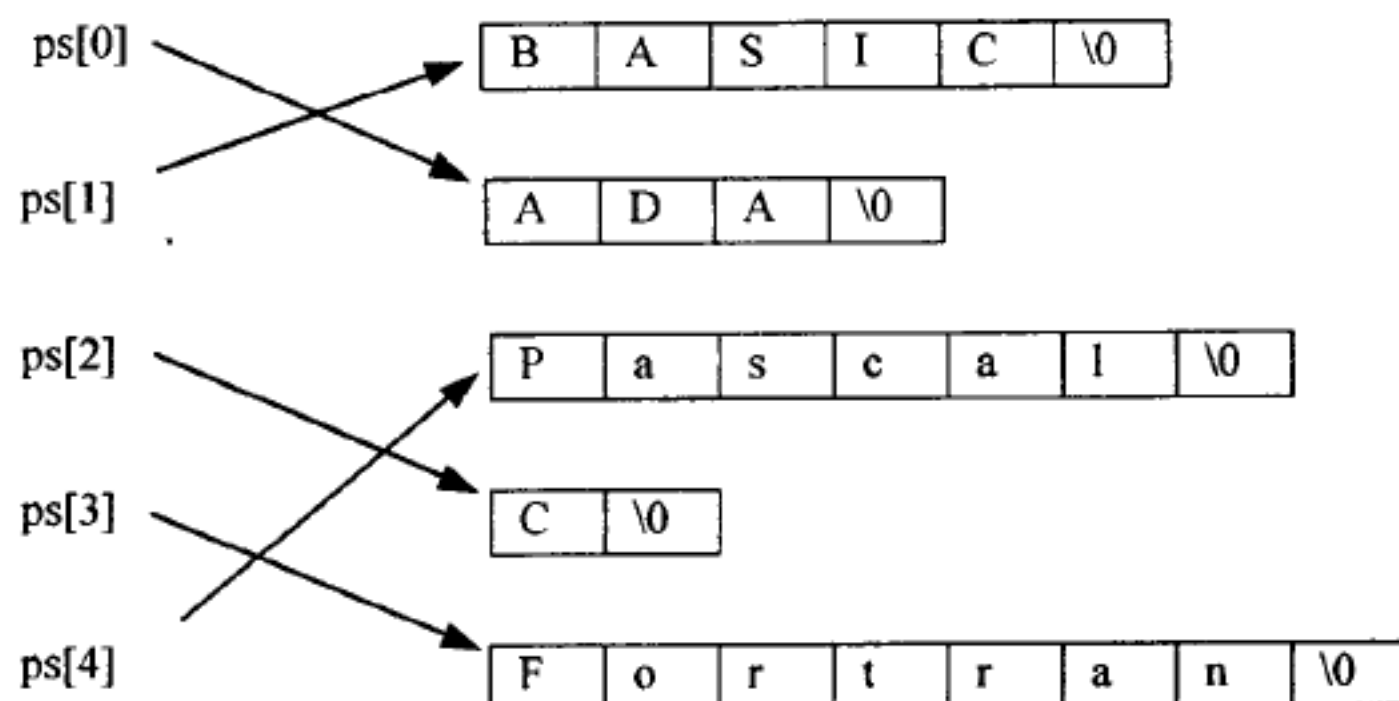
上例程序利用选择法排序字符串,程序中用到字符串比较函数 strcmp 来比较两个字符串的大小。比较过程中改变指针数组 ps 元素的指向,最后将使得元素 ps[i]的下标 i 值越小,则 ps[i]指向值越小的字符串。比较完成后,ps 各元素的指向如图5.16所示。

5.4 指向指针的指针

5.4.1 指向指针的指针

通过一个指针变量来存取变量的值,这种方式称为“间接存取”方式。例如:

```
double d, * p=&d;
```

图5.16 排序后指针数组 `ps` 的元素指向字符串数组的情况

指针变量 `p` 中存放的是变量 `d` 的地址,通过变量 `p` 可以间接地存取变量 `d` 的数据,先通过变量 `p` 得到变量 `d` 的地址,然后再存取变量 `d` 的值,这种通过一次“间接存取”就能存取变量的值,称为“一级间址”方式,如图5.17(a)所示。如果一个指针变量中存放的是另一指针变量的地址,这就需要二次“间接存取”才能存取变量的值,这称为“二级间址”,如图5.17(b)所示,这样的指针变量称为指向指针的指针。理论上,C语言的“间接存取”方式可以进行多次,构成“多级间址”,如图5.17(c)所示,但实际应用中很少超过二级间址,因为级数越高,存取速度越慢,也不易理解其存取机制。

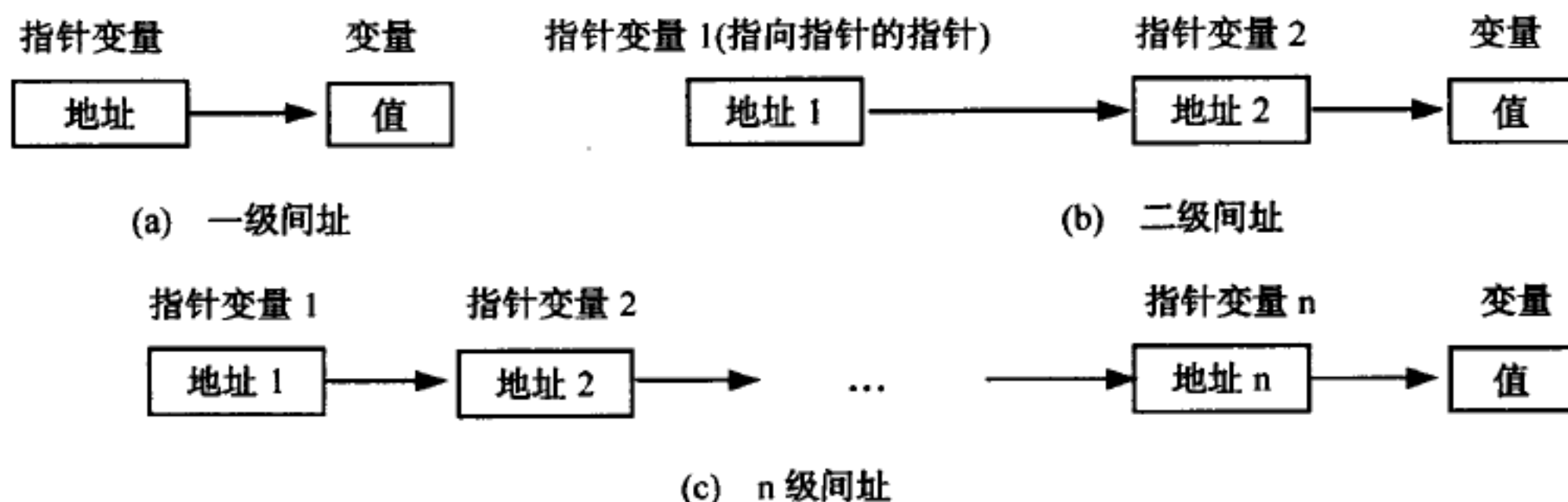


图5.17 通过指针变量存取变量的值

5.4.2 定义指向指针的指针变量

指向指针的指针变量定义的形式为:

类型名 `**` 指针变量名;

此处,指针变量名是“指向指针的指针”的变量名称,类型名是该指针变量经过二级间址后所存取变量的数据类型。由于运算符“`*`”的结合性是“从右到左”,因此“`**` 指针变量名”等价于“`*(* 指针变量名)`”,表示该指针变量的值存放的是另个指针变量的地址,要经过两次间接存取后才能存取到变量的值。例如语句:

```
double ** pp;
```


定义 pp 为指向指针的指针变量,它要经过两次间接存取后才能存取到变量的值,该变量的数据类型为 double。

5.4.3 指向指针的指针变量的应用

1. 指向一个指针变量,间接存取变量的值

可以把一个指针变量的地址赋值给指向指针的指针变量,然后通过二级间址方法存取变量的值。

[例5.14] 一个简单的例子说明如何通过二级间址方法存取变量的值。

```
main()
{
    double d=123.456, *p, **pp;
    pp=&p;
    p=&d;
    printf("d=%8.3f, ", **pp);
    **pp+=543.21;
    printf("d=%8.3f\n", d);
}
```

运行结果:

d= 123.456, d= 666.666

上述指针变量 pp 指向指针变量 p,而指针变量 p 又指向双精度实型变量 d,如图5.18所示,图中假设指针变量 p 的地址是1500,变量 d 地址是3500。此时 *pp 表示指针变量 p 的值(即变量 d 的地址),因此表达式 **pp 与变量 d 等价。

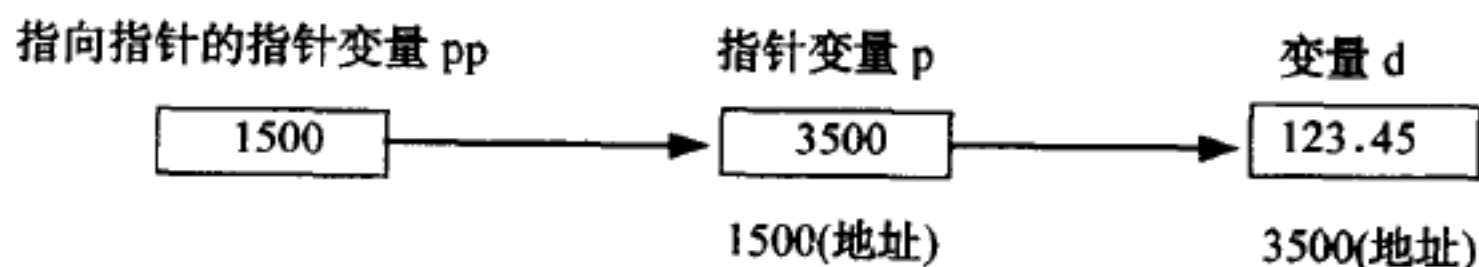


图5.18 指向指针的指针指向指针变量

2. 指向指针数组,存取指针数组元素所指内容

可以把一个指针数组的首地址赋值给指向指针的指针变量。例如:

[例5.15] 有三个等级分,由键盘输入1,屏幕显示“pass”,输入2显示“good”,输入3显示“excellent”。

```
main()
{
    int grade;
    char *ps[]={"pass", "good", "excellent"}, **pp;
    pp=ps;
    printf("请输入等级分(1~3):");
    scanf("%d", &grade);
}
```

```
    printf(" %s\n", * (pp+grade-1) );
}
```

运行结果:

请输入等级(1~3):2

good

上述程序中 pp 指向指针数组 ps 的第一个元素 ps[0], pp+1 则指向 ps 的下一个元素 ps[1], pp+2 指向 ps[2], 如图 5.19 所示。因此 *pp 就是字符串 "pass" 的首地址, *(pp+1) 则是字符串 "good" 的首地址, *(pp+2) 是字符串 "excellent" 的首地址。

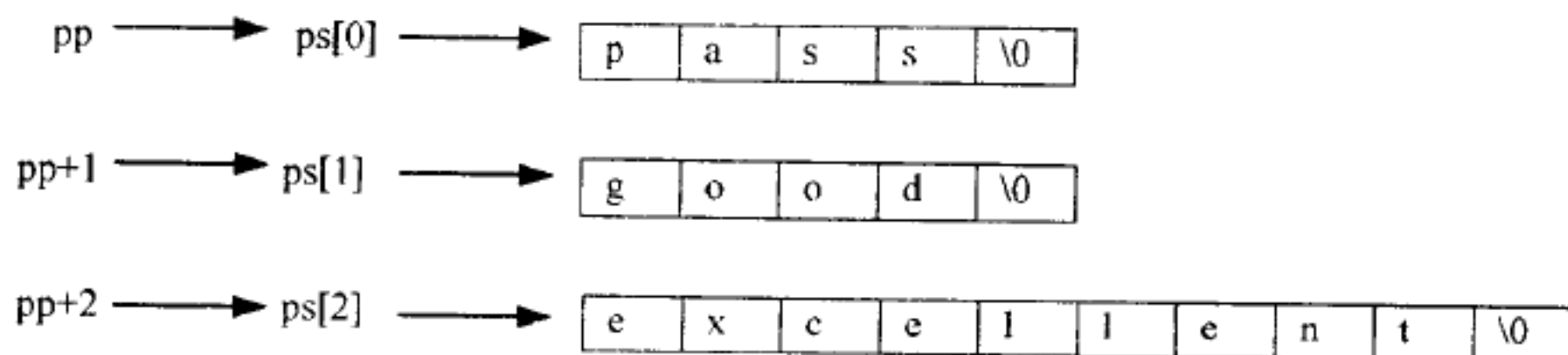


图 5.19 指向指针的指针指向指针数组

5.5 指针与结构

5.5.1 指向结构体变量的指针变量

在第四章 4.2 节已经讲述了“结构体”的概念。在定义一个结构体变量时,系统将在内存中分配一块连续的存储空间,用于存放结构体成员的数据,这块连续存储空间的首地址称为结构体变量的指针(也称为结构体变量的首地址)。可以定义指向结构体变量的指针变量,若把某结构体变量的首地址赋值给一个指针变量,则称这一指针变量指向该结构体变量。指向结构体的指针变量定义格式是:

```
struct 类型名 * 指针变量名;
```

上述类型名为结构体类型名。例如下面定义一个结构体类型 booktp 来存储书的基本信息:

```
struct booktp
{
    char name[60];    /* 书名 */
    char author[30];  /* 作者 */
    float price;      /* 价格 */
    struct datetp
    {
        unsigned year;
        unsigned month;
    } pubday;         /* 出版日期 */
}
```

```
};
```

定义了结构体类型,就可以定义结构体变量和指向结构体变量的指针:

```
struct booktp book5, *p;
```

其中 book5 为结构体变量, p 为指向结构体变量的指针。若 $p = \&\text{book5}$, 则称指针变量 p 指向结构体变量 book5, 此时可用下述三种方式之一存取结构体成员(三种方式是等价的):

1. 结构体变量名. 成员名
2. 指针变量名 \rightarrow 成员名
3. $(* \text{指针变量名}). \text{成员名}$

例如有下面程序段:

```
struct booktp *p, book5 = {"C++ Builder 网络开发实例", "清汉计算机工作室",
53, {2000, 9}};
p = &book5;
```

则 $\text{book5}. \text{price}$ 、 $p \rightarrow \text{price}$ 以及 $(*p). \text{price}$ 的值都是 53.0, 而 $\text{book5}. \text{pubday}. \text{year}$ 、 $p \rightarrow \text{pubday}. \text{year}$ 以及 $(*p). \text{pubday}. \text{year}$ 的值都是 2000。其中指向运算符“ \rightarrow ”是由两个符号“ \rightarrow ”和“ \rightarrow ”组合在一起, 就像一个箭头指向结构成员。指针运算符“ $*$ ”作用于指针变量 p 上, 构成表达式 $(*p)$, 等价于结构体变量名 book5。注意此处 $(*p). \text{price}$ 的圆括号不能少, 若写成 $*p. \text{price}$, 由于结构体成员运算符“ $.$ ”的优先级高于指针运算符“ $*$ ”, 因此 $*p. \text{price}$ 相当于 $*(p. \text{price})$, 这是一个非法的表达式。

5.5.2 指向结构体数组的指针变量

指向结构体的指针变量也可以指向结构体数组及其元素。例如, 下述程序段:

```
struct booktp *p, book[3];
p = book;
```

这样, 指针变量 p 指向结构体数组 book 的首地址, $p+1$ 指向下一个元素 $\text{book}[1]$, $p+2$ 指向元素 $\text{book}[2]$, 如图 5.20 所示, 图中假定 $\text{book}[0]$ 的地址是 3000, 由于 $\text{sizeof}(\text{struct booktp})$ 的值为 98, 每个结构体元素占内存空间 98 个字节, 因此 $p+1$ 指向地址 3098 处, $p+2$ 指向地址 3196 处。

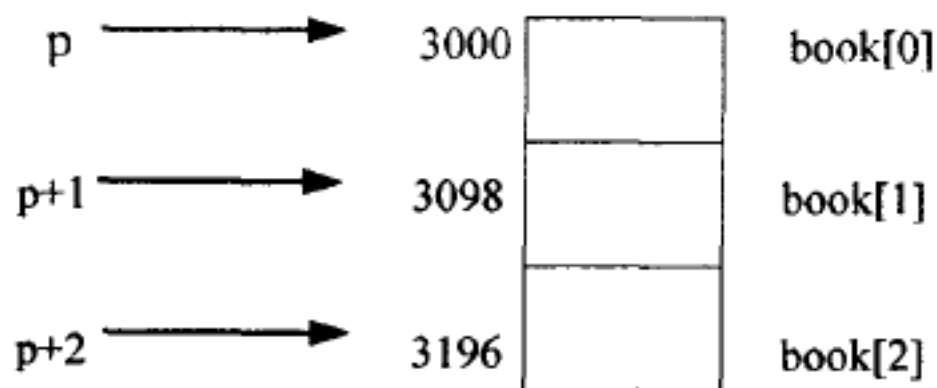


图 5.20 指针变量指向结构体数组

使用指针变量指向结构体变量或结构体数组时, 应注意运算符的优先级:

1. “ $()$ ”、“ $[]$ ”、“ \rightarrow ”、“ $.$ ”四个运算符优先级相同, 在 C 语言中具有最高的优先级, 运算的结合方向是“从左到右”;
2. “ $*$ ”、“ $++$ ”、“ $--$ ”、“ $\&$ ”四个运算符优先级相同, 在 C 语言优先级的级别仅次于上述的四个运算符, 运算的结合方向是“从右到左”。

[例5.16] 通过一个简单的例子说明指向结构体数组的指针的应用。在程序中,为了说明运算符的优先级和结合方向的用法,改变了书的价格。

```
main()
{
    struct booktp
    {
        char name[60];    /* 书名 */
        char author[30];  /* 作者 */
        float price;      /* 价格 */
        struct datetp
        {
            unsigned year;
            unsigned month;
        } pubday;        /* 出版日期 */
    };
    struct booktp *p, book[3]=
    { {"C++Builder 网络开发实例", "清汉计算机工作室", 53.0, {2000, 9}},
      {"SQL Server 循序渐进教程", "Petkovic", 35.0, {1999, 6}},
      {"VB 开发指南", "Dianne Siebold", 28.0, {2000, 9}} };
    p=book;
    printf("%8.2f,", ++p->price);
    /* 上述表达式等价于++(p->price), 即先计算++(book[0].price), 再输出 book[0].price */
    printf("%8.2f,", (++p)->price);
    /* 上述语句先计算++p, p 指向 book[1].price, 再输出 book[1].price */
    printf("%8.2f,", p++->price);
    /* 上述表达式等价于(p++)->price, 先输出 book[1].price, 再计算 p++, p 指向 book[2] */
    printf("%8.2f\n", p->price++);
    /* 上述表达式等价于(p->price)++, 先输出 book[2].price, 再计算 (book[2].price)++ */
    for(p=book; p<book+3; p++) /* 输出结构体数组的所有数据 */
        printf("%s, 作者:%s, 出版日期:%d 年%d 月, 价格:%5.1f\n", p->name,
            (*p).author, p->pubday.year, p->pubday.month, p->price);
}
```

运行结果:

54.00, 35.00, 35.00, 28.00

C++Builder 网络开发实例, 作者:清汉计算机工作室, 出版日期:2000年9月, 价格:54.0

SQL Server 循序渐进教程, 作者:Petkovic, 出版日期:1999年6月, 价格:35.0

VB 开发指南,作者:Dianne Siebold,出版日期:2000年9月,价格:29.0

上例的输出结果中,由于表达式++p->price 与 p->price++的作用,使 book[0]. price 与 book[2]. price 的值均比原初始化值多1。

5.5.3 通过指针变量存取位段数据

不能用指针变量指向位段成员,但可以用指针变量指向一个带有位段的结构体变量,此时可用下述三种方式之一存取结构体成员的值(三种方式是等价的):

1. 结构体变量名. 位段名
2. 指针变量名->位段名
3. (* 指针变量名). 位段名

[例5.17] 通过指针变量存取位段数据。

```
main()
{
    struct packed _data
    {
        unsigned a:4;
        unsigned b:3;
        unsigned c:9;
    } *p, data={ 9, 5, 60};
    p=&data;
    printf("%u,%u,%u\n", p->a, (*p).b, data.c);
    p->a+=2;
    (*p).c>>=2;
    printf("%u,%u,%u\n", p->a, (*p).b, data.c);
}
```

运行结果:

9,5,60

11,5,15

由上例知,当指针变量 p 指向带有位段的结构体变量 data 时,表达式“p->位段名”、“(*p). 位段名”与“data. 位段名”相互等价。

5.6 指向共用体和枚举型的指针

5.6.1 指向共用体变量的指针变量

指针变量可以指向一个共用体变量,此时可用下述三种方式之一存取共用体成员(三种方式是等价的):

1. 共用体变量名. 共用体成员名

2. 指针变量名—>共用体成员名

3. (* 指针变量名). 共用体成员名

[例5.18] 通过指针变量存取共用体成员数据。

```
main()
{
    struct byte _tp
    {
        unsigned char al, ah;
    };
    union reg _tp
    {
        unsigned ax;
        struct byte _tp h;
    };
    union reg _tp a, *p;
    p=&a;
    a.ax=0x3b5e;
    printf("ax=%0x, al=%0x, ah=%0x\n", p->ax, (*p).h.al, p->h.ah);
    p->h.al-=3;
    p->h.ah&=0x0f;
    printf("ax=%0x, al=%0x, ah=%0x\n", p->ax, (*p).h.al, p->h.ah);
}
```

运行结果:

ax=3b5e, al=5e, ah=3b

ax=b5b, al=5b, ah=b

上例说明了一个 reg _tp 共用体类型,以及定义相应的共用体变量 a 和指针变量 p,变量 a 占内存两个字节,可使用 a.h.ah 对它的高字节操作,使用 a.h.al 对它的低字节操作,使用 a.ax 对高、低两字节同时操作,这类似于“字存取”及“字节存取”CPU 寄存器上的数据,如图5.21所示。当指针变量 p 指向共用体变量 a 时,有下述等价表示关系。

相互等价 相互等价

p->共用体成员名 \longleftrightarrow (*p).共用体成员名 \longleftrightarrow a.共用体成员名

ax	
h.ah	h.al

高字节 低字节

图5.21 共用体 reg _tp 类型

5.6.2 指向枚举型的指针变量

指针变量可以指向一个枚举型变量,如下定义一个枚举型变量 `e` 和指向枚举型变量的指针变量 `p`:

```
enum 枚举型类型名 e, *p;
```

若 `p` 指向枚举型变量(即 `p=&e;`),则 `*p` 与变量 `e` 是等价的。

[例5.19] 通过指针变量存取枚举型变量的值。

```
main()
{
    enum fruit_type { apple, orange, banana, pineapple };
    enum fruit_type fruit[] = { orange, apple, pineapple, banana }, *p;
    char *fruit_name[] = {"苹果", "桔子", "香蕉", "菠萝"};
    for( p=fruit; p<fruit+4; p++)
        printf("%s", fruit_name[*p]);
    printf("\n");
}
```

运行结果:

桔子,苹果,菠萝,香蕉,

上例程序中 `fruit` 是一个枚举型数组,有四个元素,用指针变量 `p` 依次取出 `fruit` 数组四个元素的值:orange(对应整数1)、apple(对应整数0)、pineapple(对应整数3)、banana(对应整数2)。而 `fruit_name` 是一个指针数组,共有四个元素,分别指向四个字符串。输出时用 `*p` 作 `fruit_name` 数组的下标,此时系统自动把 `*p` 的枚举型(enum fruit_type 类型)转换成其对应的整型数据,如 `fruit_name[banana]` 等价于 `fruit_name[2]`。

5.7 指针小结

本章讲述了利用指针处理内存中各种类型数据的方法,小结如下。

5.7.1 指针概念综述

1. 变量的地址就是变量的指针。用于存储地址的变量称为指针变量。当一个变量的地址赋值给某一指针变量时,称这个指针变量指向该变量。此时,既可用变量名直接存取变量的值,也可用指针变量间接存取变量的值。

2. C语言中的数组变量、字符串数组变量、字符串、结构体变量、共用体变量、枚举型变量,甚至函数名及函数的参数(在第六章作详细介绍)以及文件(在第八章作详细介绍)等都有指针,可以定义相应的指针变量存放这些指针。同样有两种方法存取变量的值:用变量名直接存取或用指针变量间接存取。也有两种方法调用函数:用函数名来调用或用指向函数的指针变量来调用(见第六章)。

3. 指针运算符“*”作用在变量的地址上,即表达式“*变量的地址”相当于间接存取该变量的值。

4. 一维数组名是该数组的首地址(第一个元素的地址)。当指针变量 p 指向数组的某一个元素时, p+1 指向下一个元素, p-1 指向上一个元素。

5. 字符串可以存放在字符数组中,也能以字符串常量的形式出现在程序中。程序中把一个字符串常量赋值给一个指针变量,实际上是把存放该字符串常量的内存单元首地址赋值给指针变量。

6. 可以把 C 语言的二维数组 a 视为一个一维数组(a[0],a[1],a[2],...),而这个一维数组的每一个元素 a[i] 又是一个一维数组(a[i][0],a[i][1],a[i][2],...)。因此,&a[i][j]、a[i]+j 与 *(a+i)+j 三者相互等价,都是元素 a[i][j] 的地址。一个行指针变量 pi 指向的数据类型是一个有 N 个元素的一维数组,当 pi 指向二维数组的一行(设每行也有 N 个元素)时,pi+1 指向下一行,pi-1 指向上一行。

7. 指针数组的每一个元素都是一个指针变量,指针数组的元素可用来指向变量、数组元素、字符串等。

8. 指向指针的指针要进行两次“间接存取”(二级间址)才能存取变量的值。

9. 通过指针变量存取结构体变量成员数据有两种方法:一种是通过指针运算符“*”,另一种是通过指向运算符“->”。指针变量存取共用体变量的数据也与之类似。

10. 在 C 程序中使用指针编程,可以写出灵活、简练、高效的好程序,实现许多用其他高级语言难以实现的功能。

11. 初学者利用指针编程较易出错,而且这种错误往往是隐蔽的,难以发现。比如由于未对指针变量 p 赋值就对 *p 赋值,新值就代替了内存中某单元的内容,可能出现不可意料的错误。因此使用指针编程,概念要清晰,并注意积累经验。

5.7.2 指针运算小结

1. 指针变量赋值

通过如下程序段来说明指针变量的赋值:

```
int i, j, a[10], b[5][9], *p, *q, (*pi)[9], *pa[5];
char *ps;
char *pas[]={ "abc", "defghij", "kn" };
/* 将三个字符串常量"abc", "defghij", "kn" */
/* 的首地址分别赋值给指针数组 pas 的三个元素 pas[0]、pas[1]、pas[2] */
p=&i;      /* 将一个变量地址赋给指针变量 p */
q=p;      /* p 和 q 都是指针变量,将 p 的值赋给 q */
q=NULL;   /* 将 NULL 空指针赋值给指针变量 q */
p=(int *)malloc(5 * sizeof(int));
/* 在内存中分配10个字节的连续存储单元块, */
/* 并把这块连续存储单元的首地址赋值给指针变量 p */
p=a;      /* 将一维数组 a 首地址赋给指针变量 p */
p=&a[i];   /* 将一维数组 a 的第 i 个元素地址赋给指针变量 p */
```

```

p=&b[i][j];
/* 此赋值语句等价于 p=b[i]+j;或 p=* (b+i)+j;,这三条语句都是
将二维数组元素 b[i][j]地址赋给指针变量 p */
pi=b+i;
/* 此赋值语句等价于 pi=&b[i];,表示行指针 pi 指向二维数组 b 的第 i 行 */
for(i=0; i<5; i++) pa[i]=b[i];
/* 指针数组 pa 的元素 pa[i](i=0, 1, ..., 4),分别指向二维数组 b 的第 i 行的第 0
列元素 */
ps="Hello!"; /* 将字符串常量"Hello!"的首地址赋值给指针变量 ps */

```

2. 指针变量加(减)一个整数

若有下述程序段:

```

类型名 *p, *q;
p=p+n;
q=q-m;

```

此处“类型名”是指针变量 p、q 所指向变量的数据类型,并假定 m、n 为正整数,则系统将自动计算出:

p 指针向高地址方向位移的字节数 = sizeof(数据名) * n;

q 指针向低地址方向位移的字节数 = sizeof(数据名) * m;

指针变量每增1、减1一次所位移的字节数等于其所指的数据类型的大小,而不是简单地把指针变量的值加1或减1。另外,上述指向二维数组 b 的行指针 pi, pi++则指向二维数组 b 的下一行。

3. 两指针变量相比较

指向同一块连续存储单元(通常是数组)的两个指针变量可以进行关系运算。假设指针变量 p、q 指向同一数组,则可用关系运算符“<”、“>”、“>=”、“<=”、“==”、“!=”进行关系运算。若 p==q 为真,则表示 p、q 指向数组的同一元素;若 p 指向地址较大元素, q 向地址较小的元素,则 p>q 的值为1(真)。如果 p 和 q 不指向同一数组,则比较无意义。

任何指针变量或地址都可以与 NULL 作相等或不相等的比较。

4. 两指针变量相减

指向同一块连续存储单元(通常是数组)的两个指针变量可以进行相减运算。假设指针变量 p、q 指向同一数组,则 p-q 的值等于 p 所指对象与 q 所指对象之间的元素个数,若 p>q 取正值, p<q 取负值。

5.7.3 等价表达式

用指针变量指向某一数据类型的变量时,就可以通过指针来存取该变量的值。表5.3列出了通过指针存取变量值的相互等价表达式,表中 N、M 为整型常量;并假设表中所有构造型数据类型均已正确定义。此处,两个表达式“相互等价”仅仅是指“存取变量值”时两个表达式是等价的。

表5.3 通过指针存取变量值的相互等价表达式

定 义	等价条件	等价表达式	说 明
int a;		a, * &a	"*" 与 "&" 相互抵消
int * p;		p, &* p	"&" 与 "*" 相互抵消
int a, * p;	p=&a;	a, * p	指针与变量
int a[N];		a[i], *(a+i)	一维数组名与其元素
int a[N], * p;	p=a;	a[i], *(p+i), p[i]	指针与一维数组
int b[N][M];		b[i][j], *(b[i]+j), (*(b+i)+j), (*(b+i))[j], &b[0][0]+M*i+j)	二维数组名与其元素
int b[N][M], * p;	p=&b[0][0];	b[i][j], *(p+M*i+j), p[M*i+j]	指针与二维数组
int b[N][M], (* pi)[M];	pi=b;	b[i][j], *(pi[i]+j), (*(pi+i)+j), (*(pi+i))[j], pi[i][j]	行指针与二维数组
int b[N][M], * pa[N];	pa[i]=b[i]; 其中 i=1,...,N	b[i][j], *(pa[i]+j), (*(pa+i)+j), (*(pa+i))[j], pa[i][j]	指针数组与二维数组
int a, * p, ** pp;	p=&a; pp=&p;	a, * p, ** pp	指向指针的指针
struct book _tp book, * p;	p=&book;	book. 成员名, p->成员名, (*p). 成员名	指针与结构体
union reg _tp r, * p;	p=&r;	r. 成员名, p->成员名, (*p). 成员名	指针与共用体
enum furit _type f, * p;	p=&f;	f, * p	指针与枚举型

习 题

一、选择题(每题只有一个正确答案)

5.1 若已定义: int * p, a;, 则语句 p=&a; 中的运算符"&"的含义是【1】。

【1】 A)位与运算 B)逻辑与运算 C)取指针内容 D)取变量地址

5.2 若已定义: int a, * p=&a;, 则下列函数调用中错误的是【2】。

【2】 A)scanf("%d", &a); B)scanf("%d", p);
C)printf("%d", a); D)printf("%d", p);

5.3 若已定义 int a=5;, 对(1) int * p=&a; 和(2) * p=a; 两个语句的正确解释是【3】。

【3】 A)语句(1)和(2)中的 * p 含义相同, 都表示给指针变量赋值
B)语句(1)和(2)的执行结果都是把变量 a 的地址赋给指针变量 p
C)语句(1)是在对 p 进行说明的同时进行初始化, 使 p 指向 a; 语句(2)是将变量 a 的值赋给指针变量 p
D)语句(1)是在对 p 进行说明的同时, 使 p 指向 a; 语句(2)是将变量 a 的值赋给指针变量 p 所指变量

5.4 C 语言中 NULL 表示【4】。

【4】 A)空指针 B)未定义的变量

C)字符串的结束符

D)文件的结束符

5.5 若有定义 `char *p, ch;`, 则不能正确赋值的语句组是【5】。

- 【5】 A) `p=&ch; scanf("%c", p);`
 B) `p=(char *)malloc(1); *p=getchar();`
 C) `*p=getchar(); p=&ch;`
 D) `p=&ch; *p=getchar();`

5.6 下面程序段动态分配一个整型存储单元, 单元的地址给 `s`, 请选择正确答案填空。

```
int *s;
s=【6】malloc( sizeof(int));
```

- 【6】 A) `int *` B) `int` C) `(int *)` D) `void *`

5.7 下面程序段的功能是【7】。

```
char str1[300], str2[300], *s=str1, *t=str2;
gets(s); gets(t);
while(( *s)&&( *t)&&( *t== *s))
{
    t++;
    s++;
}
printf("%d\n", *s- *t);
```

- 【7】 A) 输出两个字符串长度之差
 B) 比较两个字符串, 并输出其第一个不相同字符的 ASCII 码的差值
 C) 把 `str2` 中的字符串复制到 `str1` 中, 并输出两个字符串长度之差
 D) 把 `str2` 字符串连接到 `str1` 之后, 并输出其第一个不相同字符的 ASCII 码的差值

5.8 若有以下定义和语句, 且 $0 \leq i < 5$, 则不能访问数组元素的是【8】。

```
int i, *p, a[]={ 1, 2, 3, 4, 5 };
p=a;
```

- 【8】 A) `* (a+i)` B) `p[p-a]` C) `p+i` D) `* (&a[i])`

5.9 以下程序段的运行结果是【9】。

```
char s[]="ABC"; int i;
for (i=0; i<3; i++) printf("%s", &s[i]);
```

- 【9】 A) ABC B) ABCABCABC C) AABABC D) ABCBCC

5.10 把字符串 "Ok!" 赋值给变量, 不正确的语句或语句组是【10】。

- 【10】 A) `char a[]="Ok!";` B) `char a[8]={'O','k','!','\0'};`
 C) `char *p;` D) `char p;`
 `p="Ok!";` `strcpy(p, "Ok!");`

5.11 下面程序段的输出结果为【11】。

```
float y=0.0, a[]={2.0, 4.0, 6.0, 8.0, 10.0}, *p;
int i;
p=&a[1];
for(i=0; i<3; i++) y+=*(p+i);
printf("%f\n", y);
```

【11】A)12.0000 B)28.0000 C)20.0000 D)18.0000

5.12 下面程序段的输出结果为【12】。

```
char b[]="ABCD";
char *chp=&b[3];
while(chp>&b[0])
{ putchar(*chp);
  --chp;
}
```

【12】A)DBCA B)DCB C)CB D)CBA

5.13 下面程序段的输出结果为【13】。

```
char s*="Morning", *p=s;
while(*p!='\0') p++;
printf("%d\n", (p-s));
```

【13】A)0 B)7 C)8 D)9

5.14 以下程序段的运行结果是【14】。

```
char c[]={ 'a', 'b', '\0', 'c', '\0' };
printf("%s\n", c);
```

【14】A)ab c B)'a''b' C)abc D)ab

5.15 以下各语句中,字符串"abcde"能正确赋值的操作是【15】。

【15】A)char s[5] = { 'a','b','c','d','e' };
 B)char *s; s = "abcde";
 C)char *s; gets(s);
 D)char *s; scanf("%s", s);

5.16 下面程序段的输出结果是【16】。

```
char s[10], *sp = "HELLO";
strcpy(s, sp);
s[0] = 'h'; s[6] = '!';
puts(s);
```

【16】A)hELLO B)HELLO C)hHELLO! D)h!

5.17 以下程序段的运行结果是【17】。

```
char *s="0123214";
int v1=0, v2=0, v3=0;
while(*s)
{ switch(*s)
{ default : v3++;
  case '1' : v1++; break;
  case '2' : v2++;
}
s++;
}
```

```
printf(" %d,%d,%d\n", v1, v2, v3);
```

【17】A) 5, 2, 3 B) 2, 2, 3 C) 5, 5, 3 D) 1, 0, 1

5.18 若有以下定义和语句, 值为9的正确表达式是【18】。

```
int a[][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }, *p;
p = &a[0][0];
```

【18】A) * (p + 2 * 3 + 2) B) * (* (p + 2) + 2)
C) p[2][2] D) * (p[2] + 2)

5.19 若有定义 `int a[3][5], i, j;` (且 $0 \leq i < 3, 0 \leq j < 5$), 则 `a[i][j]` 的地址不正确表示是【19】。

【19】A) &a[i][j] B) a[i] + j
C) * (a + i) + j D) * (* (a + i) + j)

5.20 若有以下定义和语句, 则对 `a` 数组元素地址的正确引用是【20】。

```
int a[2][3], (*p)[3];
p = a;
```

【20】A) * (p + 2) B) p[2] C) p[1] + 1 D) (p + 1) + 2

5.21 若有定义 `int *p[4];`, 则标识符 `p` 是一个【21】。

【21】A) 指向整型变量的指针变量
B) 指向函数的指针变量
C) 指向有四个整型元素的一维数组的指针变量
D) 指针数组名, 有四个元素, 每个元素均为一个指向整型变量的指针

5.22 若有定义 `int (*ptr)[M];`, 则标识符 `ptr` 是【22】。

【22】A) M 个指向整型变量的指针
B) 指向 M 个整型变量的函数指针
C) 一个行指针, 指向有 M 个整型元素的一维数组
D) 有 M 个指针元素的一维指针数组, 每个元素都只能指向整型变量

5.23 设有如下定义, 则以下说法中不正确的是【23】。

```
char *ps[2] = { "abc", "ABC" };
```

【23】A) `ps` 为指针变量, 它指向含有两个数组元素的字符型一维数组
B) `ps` 为指针数组, 其两个元素中各自存放了字符串 "abc" 和 "ABC" 的首地址
C) `ps[1][2]` 的值为 'C'
D) `* (ps[0] + 1)` 的值为 'b'

5.24 下面程序段的运行结果是【24】。

```
int a[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
int *p[4], j;
for (j = 0; j < 4; j++) p[j] = a[j];
printf(" %2d, %2d, %2d, %2d\n", *p[1], (*p)[1], p[3][2], * (p[3] + 1) );
```

【24】A) 4, 4, 9, 8 B) 程序出错
C) 4, 2, 12, 11 D) 1, 1, 7, 5

5.25 若有定义 `char *language[] = { "FORTRAN", "BASIC", "PASCAL", "JAVA", "C" };`, 则 `language[2]` 的值是【25】。

【25】A)一个字符 B)一个地址 C)一个字符串 D)不定值

5.26 设有以下定义,则下列能够正确表示数组元素 $a[1][2]$ 的正确表达式是【26】。

```
int a[][3]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

```
int *p=a[0];
```

【26】A) $*((*p+1)[2])$

B) $*(* (p+5))$

C) $(*p+1)+2$

D) $*(* (a+1)+2)$

5.27 设有以下语句,则值为11的表达式是【27】。

```
struct st
```

```
{
```

```
    int n;
```

```
    char ch;
```

```
} a[3]={10, 'A', 20, 'B', 30, 'C'}, *p=&a[0];
```

【27】A) $p++->n$

B) $p->n++$

C) $(*p).n++$

D) $++p->n$

5.28 下面程序段的运行结果是【28】。

```
struct stu
```

```
{ int x, int *y;
```

```
} *p;
```

```
int dt[]={ 1, 2, 3, 4 };
```

```
struct stu a[4]={ 5, &dt[0], 6, &dt[1], 7, &dt[2], 8, &dt[3] };
```

```
p=a;
```

```
printf( "%d," , (++p)->x );
```

```
printf( "%d," , ++p->x );
```

```
printf( "%d\n" , ++(*p->y) );
```

【28】A) 6,7,3

B) 6,6,3

C) 6,6,2

D) 5,7,2

5.29 阅读以下程序,当程序运行时输入 hi 后回车,则程序输出 6968;若程序运行时输出为 6869,那么输入的是【29】。

```
#include <stdio.h>
```

```
main()
```

```
{ union val
```

```
    { char cval;
```

```
      int ival;
```

```
    } x;
```

```
char *p;
```

```
p=&x.cval+1;
```

```
scanf( "%c%c" , &x.cval, p );
```

```
printf( "%x\n" , x.ival );
```

```
}
```

【29】A) lp

B) **

C) jn

D) ih

5.30 以下程序段的输出结果是【30】。

```
union u _tp
```

```

{ char ch[2];
  int a;
} r={'A'}, *p=&r;
p->ch[1]='B';
printf("%x\n", p->a);

```

【30】A)4142

B)4241

C)AB

D)BA

二、填空题

5.31 下面程序按逆序重新放置 array 数组中 N 个元素的值, array 数组中的值由用户输入。

```

#include <stdio.h>
#define N 10
main()
{ int array[N], i, *ps, *pe, temp;
  for( i=0; i<N; i++ ) scanf( "%d", array+【1】 );
  for( ps=array, pe=array+N-1; ps<pe; ps++, 【2】 )
  { temp=*ps;
    *ps=【3】;
    *pe=temp;
  }
  for( i=0; i<N; i++ ) printf( "%d ", array[i] );
  printf( "\n" );
}

```

5.32 下面程序是实现从主串 str 中取出一子串 sub, n 表示取出的起始位置, m 表示所取子串的字符个数, 程序运行后输出 cdefg。

```

#include <stdio.h>
main()
{ char str[100]="abcdefgh", sub[100], *p=str, *psub=sub;
  int n=3, m=5;
  for ( p=str+n-1; p<str+【4】; p++, psub++ )
    *psub=*p;
  【5】;
  puts(sub);
}

```

5.33 下面程序用来输出字符串。

```

#include <stdio.h>
main()
{
  char *a[]={ "for", "switch", "if", "while" };
  char **p;
  for( p=a; p<a+4; p++ ) printf( "%s\n", 【6】 );
}

```


- 5.34 以下程序功能是当输入学生序号(1 或 2 或 3)后,能输出该学生的全部成绩(共有三位学生,每位学生有 4 门成绩)。

```
#include <stdio.h>
main()
{ float score[][4] = { {60,70,80,90}, {56,89,67,88},
  {34,78,90,66} }, (*p)[4];
  int n,i;
  p = 【7】;
  scanf( "%d", &n );
  printf( "序号为%d 的学生成绩是:", n );
  for( i=0; i<4; i++ ) printf( "%5.1f", p【8】 );
}
```

- 5.35 下面程序实现从字符串 str 中取出连续的数字作为一个正整数,依次存放到 a 数组中,并统计共存放了多少个正整数 n。下面程序将输出:123 456 16639 7890。

```
#include <stdio.h>
main()
{ char s[255]="ab123 x456,xy16639ghks7890# zxy", *p;
  int n=0, a[255], i, is=0;
  /* is 用于判定 *p 是否为'1',...,'9'之间的字符是,则 is=1,否则,is=0 */
  for ( p=s; p<s+strlen(s); p++ )
  { if ( !is && *p>='0' && *p<='9' )
    { is = 1;
      【9】;
      a[n-1] = 0;
    }
    if ( is && ( *p<'0' ||【10】 ) ) is = 0;
    if ( is ) a[n-1] = a[n-1]*10+*p-【11】;
  }
  for ( i=0; i<n; i++ ) printf( "%d", a[i] );
  printf( "\n" );
}
```

- 5.36 以下程序统计字符串 s 中元音字母(a、A、e、E、i、I、o、O、u、U)的个数。

```
#include <stdio.h>
main()
{
  char str[255], a[]="aAeEiIoOuU", *p, *s=str;
  int n=0;
  gets(str);
  while( *s )
  { for( p=a; *p; p++ )
```

```

        if(【12】)
        {
            n++;
            break;
        }
        【13】;
    }
    printf("字符串中元音字母的个数为:%d\n", n);
}

```

三、程序设计题(要求用指针方法完成)

- 5.37 输入 10 个整数,找出其中最大数和次最大数。
- 5.38 编写程序,交换数组 a 和数组 b 中的对应元素。
- 5.39 有10个数围成一圈,求出相邻三个数之和的最小值。
- 5.40 n 个人围成一圈,顺序排号,从第一个人开始报数(从1到 n 报数),凡报到 n 的人就从圈里出来;然后再从下一个人开始报数,直到只剩一个人为止。输出从圈里出来人的序号。
- 5.41 产生动态数组。输入数组大小后,通过动态分配内存函数 malloc 产生数组。
- 5.42 编写程序,复制字符串。
- 5.43 编写程序,连接两个字符串。
- 5.44 编写程序,将一个字符串反向存放。
- 5.45 编写程序,求字符串的长度。
- 5.46 输入一串英文文字,统计其中字母(不区分大小写)的数目。
- 5.47 由键盘输入的两个高精度正整数(不超过230位),求它们的和(精确值)。
- 5.48 对一个长度为 n 的字符串从其第 k 个字符起,删去 m 个字符,组成长度为 n-m 的新字符串,并输出处理后的字符串。
- 5.49 编写程序,输入两个字符串 str 与 substr,删除主字符串 str 中的所有子字符串 substr。
- 5.50 编写程序,实现在字符串 s1中的指定位置第 k 个字符处插入字符串 s2。
- 5.51 找出一个二维数组中的鞍点,即该位置上的元素在该行上最大,在该列上最小。有可能没有鞍点。
- 5.52 输入 n 个字符串,用指向指针的指针的方法按从大到小的顺序排列后输出。
- 5.53 输入 n 行英文字,对其进行一项英文语法检查,把每个英文句子的第一个字母改为大写。假设每个英文句子可分别由标点符号“.”、“!”或“?”结束。
- 5.54 将 n 个学生的数据包括学号、姓名、三门课成绩及总分放在一个结构体数组中,用指向指针的指针的方法,按三门课的总分从小到大的顺序排列后输出每个学生的数据。
- 5.55 桥牌使用52张扑克牌,编写一个模拟人工洗牌的程序,将洗好的牌分别发给四个人。

第 6 章

函 数

在 C 语言中,语句完成程序要执行的每一步动作,而函数则是实现程序要求的各项任务或过程。我们可以把程序组织成若干个模块,并分别用函数来实现它们,这样也就允许我们使用一条函数语句来代替一组语句。在使用函数时我们可以把函数看成一个“黑盒子”,只要将数据送进去就能得到需要的结果,而函数内部究竟是如何工作的,外部程序是不知道,也是不需要知道的,外部程序仅限于给函数输入什么以及函数输出什么(函数的界面)。函数提供了编制程序的手段,常用来把复杂的编程问题化为若干易于解决的小问题,使程序容易读、写、理解,也使得程序易于排除错误、修改和维护。

一个 C 程序中必须至少有一个函数,而且必须有一个并且仅有一个以 main 为名,这个函数称为主函数,主函数是整个程序的入口和正常的出口,正如在前面例子中所看到的那样,整个程序从主函数开始执行,也在主函数中结束。而其他函数的个数是没有限制的。C 程序的可执行部分只出现在函数的内部,粗略地看,C 程序是由一个个函数构成的,很多重要的功能也是以函数语句实现,因此我们也称 C 语言是函数语言。

一般地,我们将程序中大量基本重复的程序段以及功能相对独立的程序段用函数来实现。事实上,C 语言程序一般是由大量的小函数而不是少量的大函数构成的,即所谓“小函数构成大程序”。这样的好处是让程序各部分相互充分独立,并且任务单一,因而这些充分独立的小模块也可以作为一种固定规则的小“构件”,用来构成新的大程序。另外,C 语言的一个主要特点就是可以建立函数库,可以用所需要的库函数来编写程序完成程序的任务,如我们已经使用过的系统库函数 printf()、scanf() 等等。

本章建议课堂讲授 8 学时,上机 4~8 学时,自学 8 学时。

6.1 常见的系统库函数

C 语言的库函数是系统预先定义好的函数(或宏),它们包含许多常见且通用的初等程序功能模块,同时也覆盖了操作系统所提供的各种操作。库函数的引入,既扩大了语言本身的功能,也方便了用户的使用。一般而言,库函数都是高效而且可靠的,可以节省我们大量的时间和精力。

库函数按其功能特征分为若干类。需要注意的是:应用程序在调用 C 语言的库函数时,应当根据库函数的种类,在程序的开始处加上包含相应头文件的预处理行,例如要调用数学类库函数 sqrt(x) 时,就应当在程序开始位置加上 #include "math. h",正如我们在前面的例子中所做的那样。

本节仅介绍三类初学者常用的部分库函数,更多的内容可参阅附录 F。

6.1.1 字符与字符串函数

Turbo C 标准函数库具有许多涉及到字符和字符串的函数集。在 C 语言中,字符是一个单字节的值,而字符串是以空字符(NULL)结尾的字符数组;字符函数要求用头文件 `ctype.h` 提供它们的说明,字符串函数需要使用头文件 `string.h`,字符、字符串输入输出函数则是用 `stdio.h` 作为它们的头文件。

在 Turbo C 中一个可打印的字符也就是一个能在终端上显示的字符,它们的值在 0x20 (空格)到 0x7E(~)之间;控制字符的值在 0 至 0x1F 之间,还包括 0x7F,它们是标准 ASCII 字符集。另外还有扩展 ASCII 字符集,其码值在 0x80 至 0xFF 之间。(参见附录 D ASCII 字符集)

在字符函数中字符参数是以整型参数说明的,此时函数只用到了整型参数的低字节。类似于表达式中的字符量,在调用字符函数时,字符参数会转换为 `int` 型。

下面是一些简单常用的字符与字符串函数,学习和使用它们有利于体会函数的概念以及初步掌握系统库函数的运用。

1. `tolower()` 和 `toupper()` 函数

调用方式:`int tolower(int ch)`

`int toupper(int ch)`

说明:`tolower()` 和 `toupper()` 的原型都在头文件 `ctype.h` 中。

功能:如果 `ch` 是一个字母,函数 `tolower()` 将返回它的小写字母,函数 `toupper()` 将返回它的大写字母。如果 `ch` 不是字母则返回字符 `ch` 本身。

[例 6.1] 下面程序将字符串数组 `a` 中的每个字符串的第一个字符转换为大写字母(如果是字母的话),其他的字母都转换为小写字母。

```
#include "ctype.h"
main()
{
    char a[3][3];
    int i,j;
    for(i=0;i<3;i++) scanf("%s",a[i]);
    for(i=0;i<3;i++)
    {
        j=0;
        a[i][j]=toupper(a[i][j]);
        while(a[i][++j]) a[i][j]=tolower(a[i][j]);
    }
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++) printf("%c",a[i][j]);
        printf("\n");
    }
}
```

```
}
```

运行结果:

```
foR
iNT
l2E      /* 以上由键盘输入 */
For
Int
L2e
```

2. gets()和 puts()函数

调用方式:char * gets(char * str)

int puts(char * str)

说明:gets()和 puts()函数的原型都在头文件 stdio.h 中。

功能:gets()函数从标准输入设备(通常是键盘)读取字符并把它们放到 str 指向的字符数组中,它读取字符直至遇到换行符或文件结束符(EOF),换行符或文件结束符被转换为空字符并作为字符串的结束符,操作成功将返回 str,否则返回空指针;puts()函数把 str 指向的字符串写到标准输出设备(通常是屏幕)中去,空字符(字符串结束标志)被转换为换行符,调用成功时返回换行符(ASCII 值),失败时返回 EOF。

[例 6.2] 利用 gets()和 puts()函数输入和输出字符串。

```
#include "stdio.h"
main()
{
    char a[80], *s;
    s=gets(a);
    while(*s=='\0') s++;
    *s='!';
    * (++s)='\0';
    puts(a);
}
```

运行结果:

```
Hello      /* 键盘输入 */
Hello!
```

上例程序运行后光标将停在输出结果 Hello! 的下一行起首位置。

3. strcat()函数

调用方式:char * strcat(char * str1,char * str2)

说明:strcat()的原型在头文件 string.h 中。

功能:函数 strcat()将字符串 str2 连接到字符串 str1 上,并返回 str1。

[例 6.3]

```
#include "string.h"
main()
{
```



```

static char s1[11]={'C','\0'};
char *s,s2[]=" Program. ";
s=strcat(s1,s2);
printf("%s",s);
printf("%s",s1);
printf("%s\n",s2);
}

```

运行结果:

C Program. C Program. Program.

4. strcmp()函数

调用方式: `int strcmp(char *str1, char *str2)`

说明: `strcmp()`的原型在头文件 `string.h` 中。

功能: 该函数按先后顺序依次比较两个字符串 `str1` 和 `str2` 中的字符, 若两个字符串相等则返回数值 0, 否则返回第一对不相等字符的 ASCII 码值之差。

如函数调用 `strcmp(str1, str2)` 对于不同的字符串 `str1` 和 `str2` 有不同的返回值:

表 6.1 例举函数调用 `strcmp(str1, str2)` 的返回值

str1	str2	返回值	说明
"ABCDEFGH"	"ABCDABCEFGH"	4	'E' - 'A' = 4
"1234567"	"123"	52	'4' - '\0' = 52
"fg2BGFewr"	"fg2BGFewr"	0	
"1234567"	"987654"	-8	'1' - '9' = -8
"abc"	"abcd"	-100	'\0' - 'd' = -100

可以看出在不同的函数参数 `str1` 和 `str2` 下函数调用的返回值不同, 其含义如下:

表 6.2 `strcmp()` 的返回值含义

函数返回值	含义
小于零	字符串 <code>str1</code> 小于字符串 <code>str2</code>
等于零	字符串 <code>str1</code> 等于字符串 <code>str2</code>
大于零	字符串 <code>str1</code> 大于字符串 <code>str2</code>

5. strcpy()函数

调用方式: `char *strcpy(char *str1, char *str2)`

说明: `strcpy()`的原型在 `string.h` 中。

功能: 该函数将字符串 `str2` 的内容复制到字符串 `str1` 中, 并返回 `str1`。

[例 6.4]

```

#include "string.h"
#include "stdio.h"
main()
{
    static char s[20]={'H','e','l','l','o',' ','W','o','r','l','d','\0'};

```

```

    char *p="Hi,World!", *str;
    str=strcpy(s,p);
    puts(s);
    puts(p);
    puts(str);
}

```

运行结果:

```

Hi,World!
Hi,World!
Hi,World!

```

6. strlen() 函数

调用方式: unsigned strlen(char *str)

说明: strlen() 的原型在 string.h 中。

功能: 该函数用来计算字符串 str 的长度(字符个数,但不包含作为字符串结束标志的空字符),并返回这个长度。

表 6.3 例举 strlen(str) 函数调用

字符串 str 的定义	strlen(str) 的返回值
char str[10]={ '1','2','3','4','5','\0' };	5
char *str="abc defg";	8
char str[10]={ '\0' };	0

以上 2 至 6 项所列的是处理字符串的常用函数,由于在 C 语言中系统库函数对数组的操作不进行边界(数组大小)检查,因而在编程时使用字符串(即字符数组)函数时应确保字符数组的长度足够长,否则程序将难以正确地执行。例如在函数 strcat() 中,字符数组 str1 应足够长,这样才能够包容 str1 原来的内容以及需要连接的 str2 内容。在使用函数 strcpy() 时也应注意使字符数组 str1 的长度不短于 str2 的长度等。另外,对于字符串的使用还应注意字符串结束符(空字符'\0'),虽然它不是字符串的内容却是字符串的重要组成部分,没有它字符数组将不成为字符串。因此在使用字符串函数时应注意函数对各字符串结束符的处理。

6.1.2 简单数学函数

Turbo C 库定义了一些数学函数,我们可以利用它们方便地进行数学运算。这些函数大致可以分成以下几个种类:

- 三角函数
- 双曲函数
- 指数与对数函数
- 其他函数

所有这些数学函数的原型都包含在头文件 math.h 当中,因此用到这些数学库函数的程序时都应包含这个头文件。绝大部分数学库函数的返回值都是双精度(double)类型的,这样可以保证数学运算的精度。这里只介绍其中几个简单的函数。

1. $\sin()$ 、 $\cos()$ 及 $\tan()$ 函数调用方式: $\text{double sin(double arg)}$ $\text{double cos(double arg)}$ $\text{double tan(double arg)}$ 功能:函数 $\sin()$ 、 $\cos()$ 及 $\tan()$ 分别用于计算弧度参数 arg 的正弦、余弦及正切值。

[例 6.5] 下面程序显示从 -1 到 1 以 0.1 为增量递增的值的正弦、余弦及正切值。

```

#include "math.h"
#include "stdio.h"
main()
{
    double v=-1.0;
    while((v <= 1.0)
    {   printf("arg: %f, sin= %f, cos= %f, tan= %f\n", v, sin(v), cos(v), tan(v));
        v+=0.1;
    }
}

```

2. $\exp()$ 和 $\text{pow}()$ 函数调用方式: $\text{double exp(double a)}$ $\text{double pow(double b, double c)}$ 功能: $\exp()$ 函数返回以自然数 e 为底,函数参数 a 为幂的指数值 e^a ; $\text{pow}()$ 函数则是返回 b 的 c 次幂值 b^c 。3. $\log()$ 和 $\log_{10}()$ 函数调用方式: $\text{double log(double x)}$ $\text{double log}_{10}(\text{double } x)$ 功能:函数 $\log()$ 返回函数参数 x 的自然对数值;而函数 $\log_{10}()$ 则返回以 10 为底的 x 的对数值 $\log_{10}x$ 。

例如:函数表达式 $\exp(1.0)$ 的值为 2.718282($e \approx 2.718282$),而 $\exp(2.0)$ 等于 7.389056, $\text{pow}(2.0, 7.0)$ 等于 128.0 以及 $\text{pow}(10.0, 3.0)$ 等于 1000.0。数学上我们知道自然对数函数 $y = \log(x)$ 是指数函数 $y = \exp(x)$ 的反函数,而对数函数 $y = \log_{10}(x)$ 是幂函数 $y = \text{pow}(10, x)$ 的反函数,自然地函数表达式 $\log(2.718282)$ 的值为 1.0, $\log(7.389056)$ 等于 2.0 以及 $\log_{10}(1000.0)$ 等于 3.0 等等。当然在这里存在着一定的误差,事实上,利用计算机进行科学计算,精度是我们必须面对的问题。

数学函数都有其定义域,如对数函数 $\ln x$ 或 $\log_{10}x$ 的参数 x 不能小于或等于零等等。C 语言中数学库函数是用来计算这些数学函数的,因此在编写程序时应保证函数参数在定义域内,否则程序运行一定会出错;另外库函数在设计时有时也会因技术上的考虑加上一些人工的限制,如幂函数的 b^c 底 b 不能为零及幂 c 不能小于或等于零等等,编程时我们同样也要遵循这些限制。

4. $\text{sqrt}()$ 函数调用方式: $\text{double sqrt(double num)}$ 功能:该函数返回函数参数 num 的平方根。

例如 $\text{sqrt}(9.0)$ 等于 3.0, $\text{sqrt}(2.0)$ 等于 1.414214 等等。

5. fabs() 和 abs()、labs() 函数

调用方式: `double fabs(double num)`

`int abs(int num)`

`long labs(long num)`

功能: 这些函数均返回函数参数 `num` 的绝对值, `fabs()`、`abs()` 和 `labs()` 分别适用于浮点型(双精度型)、整型和长整型数的绝对值计算。

例如 $\text{fabs}(-9.0)$ 等于 9.0, $\text{abs}(9)$ 等于 9, $\text{labs}(-99999)$ 等于 99999。

6. fmod() 函数

调用方式: `double fmod(double x, double y)`

功能: 该函数返回函数参数之商 x/y 的余数。

表 6.4 例举 `fmod(x,y)` 函数调用

函数调用	函数返回值	说 明
<code>fmod(100.0, 3.0)</code>	1.0	$10.0 = 3.0 * 33 + 1.0$
<code>fmod(100.0, -3.0)</code>	1.0	$10.0 = (-3.0) * (-33) + 1.0$
<code>fmod(13.657, 2.333)</code>	1.992	$13.657 = 2.333 * 5 + 1.992$
<code>fmod(13.657, -2.333)</code>	1.992	$13.657 = (-2.333) * (-5) + 1.992$

比较 `fmod()` 函数与取模运算符(`%`), 不难看出取模运算表达式 $x\%y$ 的值也是 x/y 的余数, 但参与运算的 `x` 和 `y` 必须是整型量。

6.1.3 类型转换函数

程序设计中常需要处理“数字串”, 有时将它作为字符串来加工, 有时使用该“数字串”所代表的数值。类型转换函数将提供帮助, 它们的函数原型都包含在头文件 `stdlib.h` 中。

1. atoi()、atol() 和 atof() 函数

调用方式: `int atoi(char * str)`

`long atol(char * str)`

`double atof(char * str)`

功能: 函数 `atoi()`、`atol()` 或 `atof()` 将函数参数 `str` 所指向的字符串分别转换成整型(`int`)、长整型(`long`)或双精度实型(`double`)值, 并返回这个值。在 `str` 中应包含一个有效的整型、长整型或实型数, 如果不是这样, 则函数返回值为 0。

在函数参数 `str` 所指向的(数值)字符串中的数值可以以任何字符结尾, 但不能是整型(或实型)数的有效部分。例如函数调用 `atoi("12.3")` 将返回整数值 12, `atof("1.2ABC")` 将返回实数值 1.2。

2. itoa() 和 ltoa() 函数

调用方式: `char * itoa(int num, char * str, int radix)`

`char * ltoa(long num, char * str, int radix)`

功能: 函数 `itoa()` 或 `ltoa()` 将整型或长整型数 `num` 转换成与其等价的字符串, 且将其结果放在由 `str` 所指向的字符串中。字符串输出的进制由参数 `radix` 确定, 它可以在 2 到 36 之间的范

围内变化。函数返回 `str` 所指向的指针。`str` 所指向的字符串应有足够的长度来保存转换后的结果。

6.2 用户自定义函数

用户自定义函数真正体现 C 语言函数的价值,它使得用户可以模块化地组织应用程序,提高程序设计的效率和保障程序的可靠性。用户自定义函数必须首先从函数的定义入手,要合理、正确地安排函数,才能有效而充分地利用它们。

6.2.1 函数定义、调用和说明

在上一节“常用的系统库函数”的学习中我们了解到,当需要使用一个函数时应该知道关于这个函数的:

- 函数名
- 返回值类型
- 函数参数(列表)

上述内容描述了一个函数的“外形”,它们构成了该函数的接口信息,用以保证函数的定义与调用之间的一致性。在对主函数的学习中,我们了解到一个函数的内部构成即一个函数体的组成描述了函数功能的实现过程。这些信息可以使我们完成自定义函数的定义、说明和调用。

1. 函数定义

函数定义的格式如下:

```
返回值类型 函数名(函数形式参数列表)
{
    函数体
}
```

函数名是一个任何合法的标识符,可以使用不同的函数名标识程序中不同的函数。使用能有效反映函数所完成任务的函数名是个良好的习惯,这有助于使得抽象的函数功能具体化并提高软件的可重用性,特别是在应用程序足够复杂和庞大时。如果不能选择一个简洁的名字反映函数的功能,那可能是要求该函数完成的功能太多,最好把这种函数分成多个更小的函数。

返回值类型是指在函数调用时函数表达式所具有的数据类型,如系统库函数 `sqrt()` 的返回值类型为 `double` 等。

函数形式参数(简称形参)是一些变量,仅在函数体内使用。在函数调用时它们被初始化,接收从函数调用处传递过来的数据,这往往是函数完成任务所需要的原始数据。在函数形参列表中,各形参之间用逗号(,)分隔。

通常函数调用时,函数通过函数形参接收需要处理的原始数据,函数执行后将结果通过函数返回值或其他方式返回给调用者,实现函数的功能。

函数体(连同其上下花括弧)是一个复合语句,主要包含在函数体中需要使用的变量定义以及完成函数功能的语句。任何情况下都不能在函数体中定义另一个函数,即函数不能嵌套定义。

[例6.7] 下面程序段定义了一个函数 fmax, 实现在数组元素中寻找最大值的功能。

```
double fmax(double array[],int arrlen)
{
    double t;
    int k;
    t=array[0];
    for(k=1;k<arrlen;k++)
        if(t<array[k]) t=array[k];
    return(t);
}
```

在上述函数的定义中可以看到:fmax 是这个用户自定义函数的函数名;返回值类型是双精度型;函数有两个参数:一个双精度的数组 array 和一个整型量 arrlen(代表该数组的大小);在函数体中定义了两个变量,接下来的语句实现找出数组最大元素的功能,最后通过 return 语句把这个最大值赋给函数返回值。

上述函数定义的参数说明是现代风格的,其传统的形式是:

```
返回值类型  函数名(形参变量列表)
形参变量定义
{
    函数体
}
```

在此形式下,上述函数 fmax 的定义形式为:

```
double fmax(array, arrlen)
double array[];
int arrlen;
{
    ...
}
```

显然现代风格的函数定义更为简洁。

2. 函数调用

程序中函数调用以函数表达式语句出现。在一个函数的函数体内可以对其他函数的调用,即所谓函数的嵌套调用,也可以调用该函数本身,即所谓函数的递归调用,这些调用方式丰富了函数的层次,也加强了函数的功能。在函数调用之前我们需要了解函数的接口,即函数名、返回值类型、函数参数和函数的功能,这样才能正确地进行函数调用。

函数调用的一般形式如下:

```
函数名(函数实际参数列表)
```

调用时的函数名应与定义时的函数名完全相同,函数实际参数(简称实参)可以是常量、变量或其他表达式,它们在列表中使用逗号(,)分隔开来。实参为函数的形参(变量)初始化提供原始数据。

[例6.8] 下面两段程序均输出三个数组的最大元素值,分别不使用和使用自定义函数。

```

main()
{
    double a[10],b[10],c[4][5],t;
    int i,j;
    ...           /* 数据输入 */
    t=a[0];
    for(i=1;i<10;i++)
        if(t<a[i]) t=a[i];
    printf("1:最大值是%f\n",t);
    t=b[5];
    for(i=6;i<10;i++)
        if(t<b[i]) t=b[i];
    printf("2:最大值是%f\n",t);
    t=c[0][0];
    for(i=0;i<4;i++)
        for(j=0;j<5;j++)
            if(t<c[i][j]) t=c[i][j];
    printf("3:最大值是%f\n",t);
}

double fmax(double array[],int arrlen)
{
    double t;
    int i;
    t=array[0];
    for(i=1;i<arrlen;i++)
        if(t<array[i]) t=array[i];
    return (t);
}

main()
{
    double a[10],b[10],c[4][5];
    ...           /* 数据输入 */
    printf("1:最大值是%f\n",fmax(a,10));
    printf("2:最大值是%f\n",fmax(b+5,5));
    printf("3:最大值是%f\n",fmax(c,20));
}

```

以上只是自定义函数定义和调用的极简单的例子,通过对比可以看到在使用自定义函数后,程序变得清晰简洁了,有利于程序的调试和维护。

3. 函数说明

在函数调用之前,编译器必须知道这个函数返回值的数据类型,其道理与编译器在处理表达式中的变量时,必须先知道这个变量的数据类型是一样的。函数说明在程序中的位置应在该函数被调用的位置之前,如同变量定义的位置应在任何该变量的使用之前一样。函数说明的形式如下:

返回值类型 函数名();

在函数说明中返回值类型应与函数定义时的返回值类型是一致的,否则将产生语法错误。

例如我们在调用自定义函数 fmax 之前可以对它进行如下说明:

double fmax();

当函数的定义与函数的调用出现在同一个程序文件中,且函数的定义位于函数调用的位置之前时,可以省略函数说明,因为在函数定义中已经指明了该函数的返回值类型,如[例6.8]所示。如果函数在定义时的返回值类型为整型,也可以省略函数说明,编译器对于在函数调用之前程序没有指出其返回值类型的函数都默认为整型。

按现代风格,ANSI C 推荐使用函数原型进行函数说明。函数原型是从 C++ 中吸取来的,它不仅告诉编译器关于这个函数的返回值数据类型,还包括函数所要接收的参数的个数、类型及顺序,这样编译器可以对函数的调用进行进一步的检查,防止因函数参数类型不匹配而产生的错误。函数原型的函数说明如下:

返回值类型 函数名(函数参数数据类型列表);

在函数原型中,函数参数数据类型列表指明了该函数的参数个数、类型及顺序,在使用函数原型进行函数说明时,应至少保持返回值类型与函数定义时的一致,否则就会产生错误。在这里给出函数数据类型列表的好处就是在函数实参和形参类型不同时,编译器会进行强制类型转换,将实参类型根据 C 语言的类型转换规则进行转换,需要注意的是在类型转换时可能会损失精度(如浮点数 0.9999 转换为整型数 0),从而导致错误的结果。

用户自定义函数 fmax 的函数原型如下:

```
double fmax(double [],int);
```

有时为了方便阅读文档,在函数原型中包含了参量名,但编译器会忽略这些参量名。如:

```
double fmax(double array[],int arrlen);
```

系统库函数都有函数原型,这些函数原型分别保存在若干个头文件之中,如输入输出函数 printf()、scanf() 等的函数原型保存在 stdio.h 之中,数学函数 sin()、cos() 等的函数原型保存在 math.h 之中等等,在程序中需要使用系统库函数时,通过 #include "stdio.h" 及 #include "math.h" 等等将相应的函数原型包含在程序的开始位置,从而对这些的系统库函数进行必要的函数说明。

6.2.2 函数返回值

我们在调用函数时,很多情况下都希望函数表达式具有一个值可以直接参加程序要求的各种计算或操作,函数返回值可以实现此目的。函数返回值应该是函数内部的一个执行结果,同时它应该能被调用此函数的其他函数(主调函数)所用,这就需要在函数定义时,在函数体内使用专门的程序控制语句 return 来实现;函数返回值作为一个可变的量,它应该与变量一样,根据需要可以具有各种各样的数据类型。下面讨论有关函数返回值的几个问题。

如果需要函数返回值,就必需在函数体内进行指定,这是通过函数返回语句 return 来实现的,如[例6.7]中所见的那样。程序在执行完成 return 语句后,结束函数体的运行(忽略其后的所有语句)。return 语句的格式为:

```
return;    或:    return 表达式;
```

在前一种形式下,函数返回值是不确定的,通常在此时我们并不关心函数的返回值,而是仅用它来结束函数体的执行;在后一种形式下,函数要把表达式的值返回给调用者,即将表达式的值赋为函数返回值,并结束函数体的执行。另外,我们习惯上常把第二种形式写成:

```
return (表达式);
```

这种形式看上去可能更清晰一些。

在函数体内可以不使用 return 语句,也可以使用多条 return 语句。在前一种情况,我们不需要函数返回值,而此时函数的调用将在顺序地执行完函数体中的最后一条语句后结束。后一种情况,函数体的执行将有多个出口,而不是函数同时具有多个返回值。

[例6.9] 用户自定义函数 fact,用于计算 n 的阶乘 n! 与 (-1) 的 n 次方 $(-1)^n$ 之积。

```
#define IMAX 32767
```

```
int fact(int n)
```

```
{
```

```
    double s;
```

```
    int m;
```

```

    if(n<=0)
    {
        printf("数据错(n<=0)!\n");
        return;
    }
    for(m=1;m<=n;m++) s*=m;
    if(s>IMAX)
    {
        printf("结果数字太大!\n");
        return;
    }
    if(n%2==0) return (int)s;
    else return (int)s*(-1);
}

```

函数体中出现了四个 return 语句,前两个用于在不能正确计算结果时结束函数,后面两个 return 语句才是返回函数值。根据函数参数 n 的正负、大小以及奇偶情况,函数有可能以这四种情形结束。

函数返回值的数据类型可以是各种基本数据类型,也可以是指针类型以及构造型数据类型,相应地在 return 语句中的表达式也应与其一致,若不一致时,则以函数返回值类型为准,return 语句中的表达式类型要进行转换;若我们在定义函数时省略“返回值类型”,则函数返回值的类型被指定为默认的整型(int)。在有些场合我们根本不需要函数有返回值,可以指定返回值类型为无值类型(void),或称为空类型,例如下面自定义函数 prints 用于输出一个字符串,它并不需要函数有返回值:

```

void prints(char *str)
{
    while(*str) putchar(*(str++));
}

```

当函数返回值被指定为空类型时,禁止在函数体内使用“return (表达式);”语句,也不能在主调函数中用此函数表达式进行计算及操作,如语句“a = prints(s);”是非法的。事实上,我们无法将空类型与其他数据类型相互转换,这一点与空指针(void *)不同,空指针只是没有指向一个有效值的指针。

6.2.3 函数参数

函数参数在两个场合下出现:一个是在函数定义时,如:

```

double fmax(double array[],int arrlen)
{
    ...
}

```

其中的 array 和 arrlen 被称为函数的形参,用来在函数体内进行运算,实现各种算法;另一个场

合是在函数被调用时,如:

```
printf("1:最大值是%f\n",fmax(a,10));
```

函数调用 `fmax(a,10)` 中的 `a` 和 `10`, `a` 和 `10` 被称为函数的实参,用来为被调用函数提供计算所需的真实数据,以得到该数据下的计算结果,实现函数的功能。

函数形参以变量的形式出现,因此在函数定义时必须明确指定这些变量的类型,在函数体的执行过程中它们也可以被赋值加以改变,而这些变量的初始值来自于函数实参(在函数被调用时函数实参的值被对应地传递给函数形参),详细情况见第七章。

6.3 函数的嵌套调用及递归调用

C 语言中,所有的执行语句都只能出现在函数之中。同样,函数的调用也只能出现在某函数的函数体内。函数的调用以两种形式出现:函数的嵌套调用及函数的递归调用。

6.3.1 函数的嵌套调用

C 语言中,所有函数的定义都是互相平行和独立的,一个函数的定义不能包含另一个函数的定义,即不允许函数的嵌套定义,但函数的调用可以通过一个函数来调用另一个函数来实现,这就形成了函数的嵌套调用,C 语言不限制嵌套的个数和层数,这样我们就可以自由、合理地组织程序的模块结构。下面是一个两层嵌套的例子:

[例6.11]利用公式:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots$$

近似计算自然数 e 。近似的程度以上式右边的加法个数给出。算法按两层进行:

函数 `fac_v()` 计算 $\frac{1}{m!}$ ($m=1,2,\dots,n$);

函数 `cal_e()` 计算 $1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots + \frac{1}{n!}$, 作为 e 的近似值。

函数 `cal_e()` 调用 `fac_v()` 获得 $\frac{1}{m!}$ ($m=1,2,\dots,n$) 的值,而主函数 `main()` 则调用 `cal_e()` 得到自然数 e 的近似值。

```
#include "stdio.h"
main()
{
    double cal_e(int);
    int n;
    printf("请输入一个正整数:");
    scanf("%d",&n);
    printf("自然数 e 的近似值为%f\n",cal_e(n));
}
double cal_e(int n)
{
```



```

double fac_v(int);
double e=1.0;
while(n) e += fac_v(n--);
return (e);
}
double fac_v(int m)
{
    double v=1.0;
    while(m) v /= m--;
    return (v);
}

```

运行结果:

请输入一个正整数: 10

自然数 e 的近似值为 2.718282

在上例中每个函数(包括主函数)都很简单清晰,功能单一,整个程序执行的流程也很清楚,可用图6.1表示:

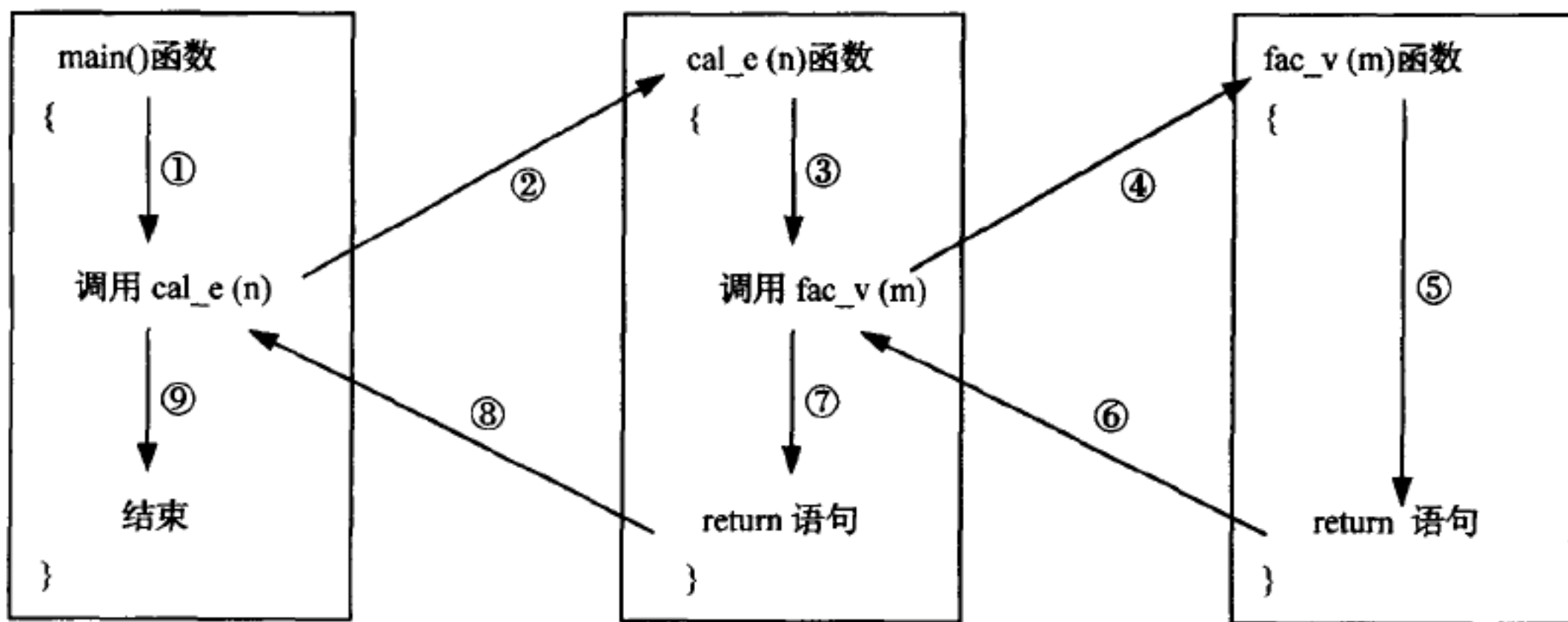


图6.1 函数嵌套的程序流程示意图

在示意图中,箭头①至⑨顺序地反映了程序流程在函数体内的走向,以及程序执行的控制权在函数之间交接的先后次序,对于更多层次的嵌套调用也只是上述两层嵌套的简单推广,多层嵌套带来的好处是可以将一个复杂的程序按自顶向下、逐步细化的思路进行设计,而在每一层的函数设计上我们可以只专注于这个函数的总体功能,而其实现的诸多细节大部分都留在下一层被嵌套的函数中去考虑,这样逐层设计可以得到清晰可靠的程序。

6.3.2 函数的递归调用

函数的递归调用是指一个函数直接或间接地调用了该函数本身,它有两种形式:

• 直接递归调用

·间接递归调用

1. 递归的形式

直接递归调用是在该函数的函数体内直接调用了它本身,如:

[例6.12] 下面函数 `sum()` 计算整数和: $1+2+3+4+\cdots+n$ 。

```
long sum(int n)
{
    if(n==1) return (1);
    return (sum(n-1)+n);
}
```

在上述函数定义的函数体内,显式地出现了对该函数本身的调用 `sum(n-1)`;而间接递归调用指被该函数嵌套调用的其他函数调用了该函数,例如将上面的 `sum()` 函数改写成两个函数 `acc()` 和 `add()` 来共同实现这个计算:

```
long acc(int n)
{
    long add(int); /* 函数说明 */
    if(n<=1) return (1);
    return (add(n));
}
```

```
long add(int n)
{
    long acc(int); /* 函数说明 */
    return (acc(n-1)+n);
}
```

在函数 `acc()` 中嵌套调用了函数 `add()`,而在函数 `add()` 中又来调用函数 `acc()`,这样就形成了间接递归调用。这里调用 `acc(n)` 和调用 `add(n)` 是等价的,功能上它们都与调用 `sum(n)` 相同:计算 $1+2+3+\cdots+n$ 。

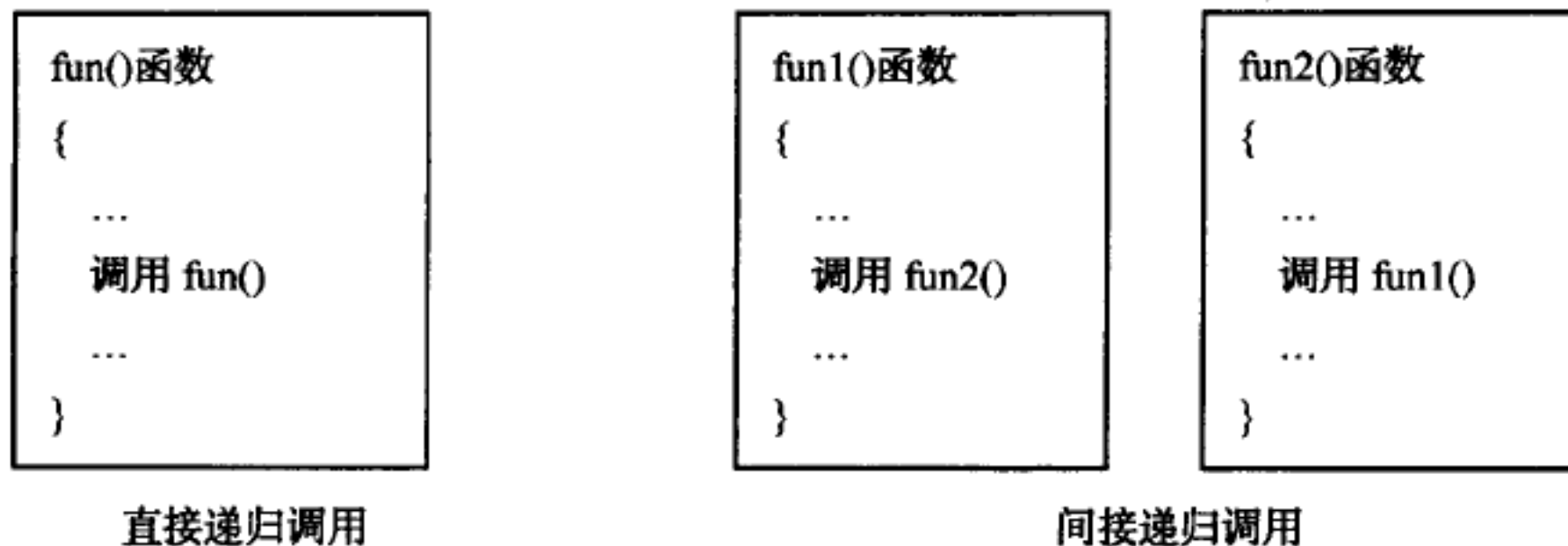


图6.2 递归调用的形式

间接递归调用并不限制嵌套的层数,上面只是两层嵌套的间接递归调用的示意图。从递归的角度看,间接递归调用与直接递归调用没有本质的差别,下面仅就直接递归调用进行讨论。

2. 递归的过程

从递归的形式可以看到,递归函数在其本身的函数体内调用了自己。从嵌套的角度看,这里包含了一个循环过程,在这个循环过程中函数进行递推的计算或操作(递推过程);从循环的角度看,任何有效的循环都应有循环条件(递归条件),或者说循环结束的出口(递归出口)。在[例6.12]中给出的 `sum()` 函数是最简单的递归函数之一,具有以上两个递归函数的必备特征:

- 递推过程:“`return (sum(n-1)+n);`”语句计算 $\text{sum}(n-1)+n$,并将它作为函数调用 `sum(n)` 的返回值,其中函数调用 `sum(n-1)` 维持循环,直至递归出口。

• 递归出口：“if(n==1) return (1);”语句在 $n=1$ 时结束递推过程(这里没有 `sum()` 函数的调用),并将数值1作为 `sum(1)` 的返回值。

需要注意,在函数的递归调用时,到达递归出口并不意味着递归过程的结束(这与循环不同),如函数调用 `sum(5)` 的递归过程中,在递归出口只得到了 `sum(1)` 的值,这还不是我们所需要的。函数调用机制将自动地进行迭代计算,最终得到 `sum(5)`,如下图所示:

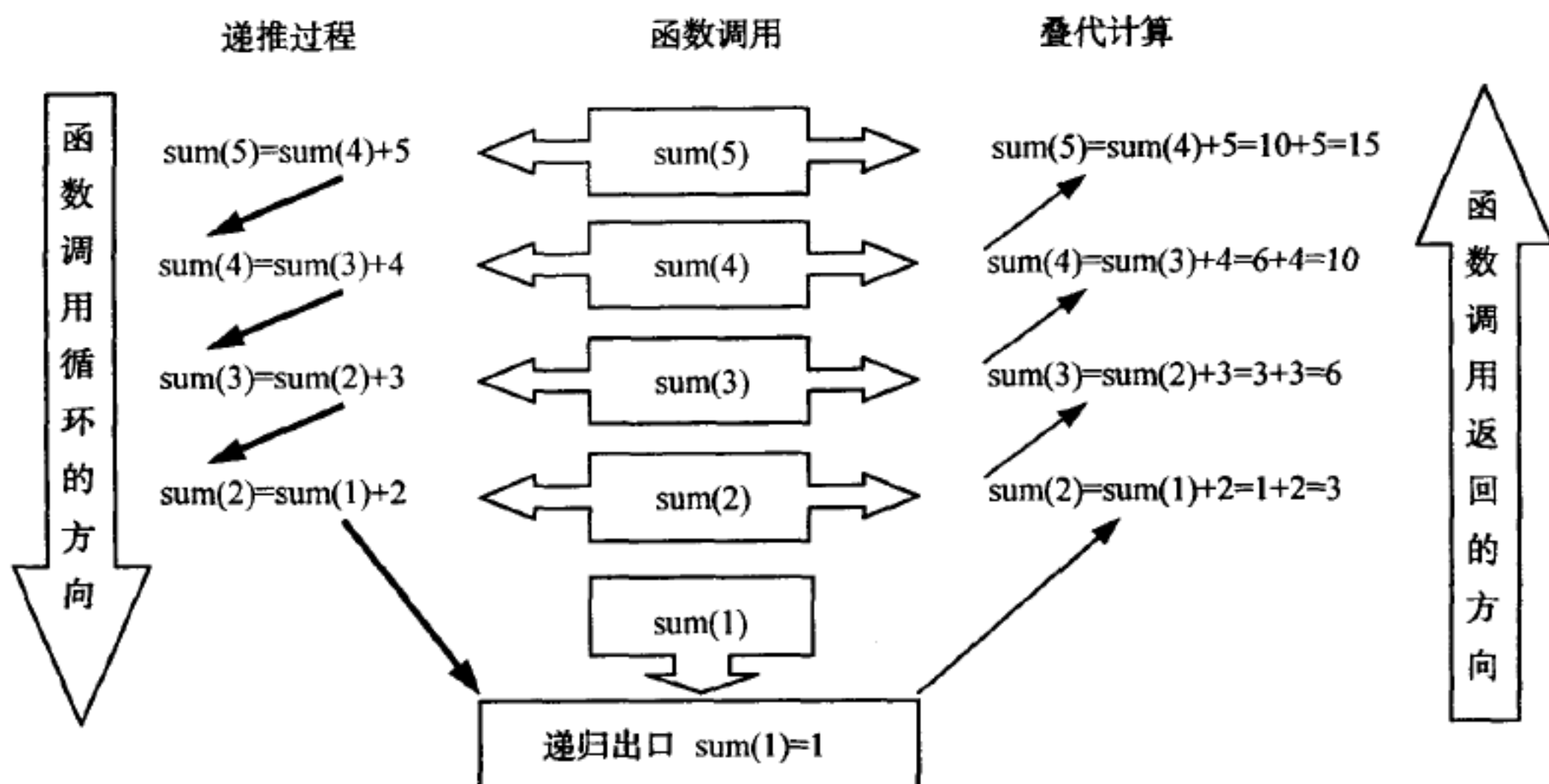


图6.3 递归函数 `sum()` 的函数调用过程示意图

这个函数调用过程:函数调用从 `sum(5)` 开始一直调用到 `sum(1)`,然后从 `sum(1)` 一直返回到 `sum(5)` 而获得了函数调用 `sum(5)` 的函数返回值。图中左边的粗箭反映了函数递归调用时的递推过程(函数参数 n 作为“循环控制变量”),右边的细箭头则反映函数返回值的迭代计算。

递归函数 `sum()` 也可以写成如下形式:

```
long sum(int n)
{
    if(n!=1) return (sum(n-1)+n);
    return (1);
}
```

此时 if 语句中的条件表达式 $n!=1$ 即为递归条件,它相当于循环中的循环条件,可以看出递归条件和递归出口在递归中起的作用是一致的。

3. 递归的算法

对于[例6.12],可以描述该求和问题为求解数学函数:

$$s(n)=1+2+3+\cdots+n$$

习惯上我们仅按求解的计算顺序看待这个问题:

$$s(n)=(1)+2+3+\cdots+n=(1+2)+3+\cdots+n=(1+2+3)+\cdots+n=\cdots=(1+2+3$$

+...+n)

即我们由始至终地求解 $s(1), s(2), \dots$, 最后得到 $s(n)$ 。

而递归的方式则不同:

$$\begin{aligned}
 s(n) &= [1+2+3+\dots] + n = \dots = [1+2+3] + \dots + n \\
 &= [1+2] + 3 + \dots + n && \text{递推关系的表示} \\
 &= [1] + 2 + 3 + \dots + n && \text{递推关系的终点/迭代求解的开始} \\
 &= ([1+2]) + 3 + \dots + n = ([1+2+3]) + \dots + n = \dots = ([1+2+3+\dots+n]) \\
 &&& \text{迭代数值求解}
 \end{aligned}$$

递归方式首先进行的是由终至始地个逐描述求解问题的递推关系, 达到最简单的情形 ($s(1)=1$), 然后就可以由递推关系从始至终地迭代求解数值了, 经历了这样一个由一般到特殊再由特殊到一般的辩证过程。递归函数 $s(n)$ 具有以下性质:

- 递推关系: $s(m) = s(m-1) + m, m=1, 2, \dots, n$
- 极端情形: $s(1)=1$

了解了待求解问题的递归性质, 就不难对应地写出相应的递归函数, 递推过程即为递推关系的算法表示, 而递推出口是极端情形的程序语言描述, 完成了这两点, 就完成了递归函数的设计。

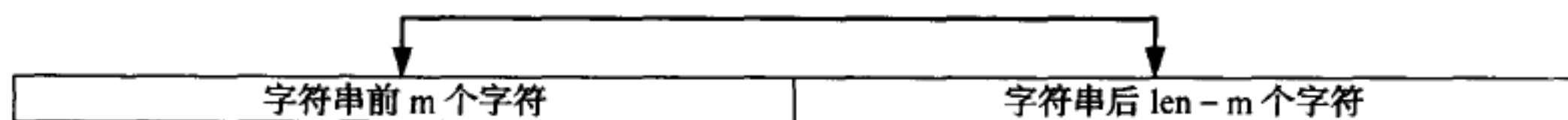
递归算法对许多复杂问题具有简洁的描述能力。对比一般的算法和递归算法, 可以发现一般的算法通常是从初始条件到最终结果的过程, 而该过程往往没有统一的模式或规范可以遵循, 因而可能是五花八门且结构复杂的; 而递归算法则强调求解的方法描述, 用以构造求解的过程, 并通常是将一个大规模问题转化为同类小规模问题的简单模式进行处理, 因而程序具有天然的简洁、紧凑、可读性好等优点。

关于递归函数设计的一个重要问题是有效性。一个有效的递归函数应能在有限步循环调用达到递推出口, 否则对该函数的调用将是一个无限过程(无限递归)并耗尽内存空间等系统资源, 在编写和运行递归函数时应注意这个问题(一个正确的递归函数可能因为输入不正确导致无限递归, 如[例6.12]的函数调用 $\text{sum}(0)$)。

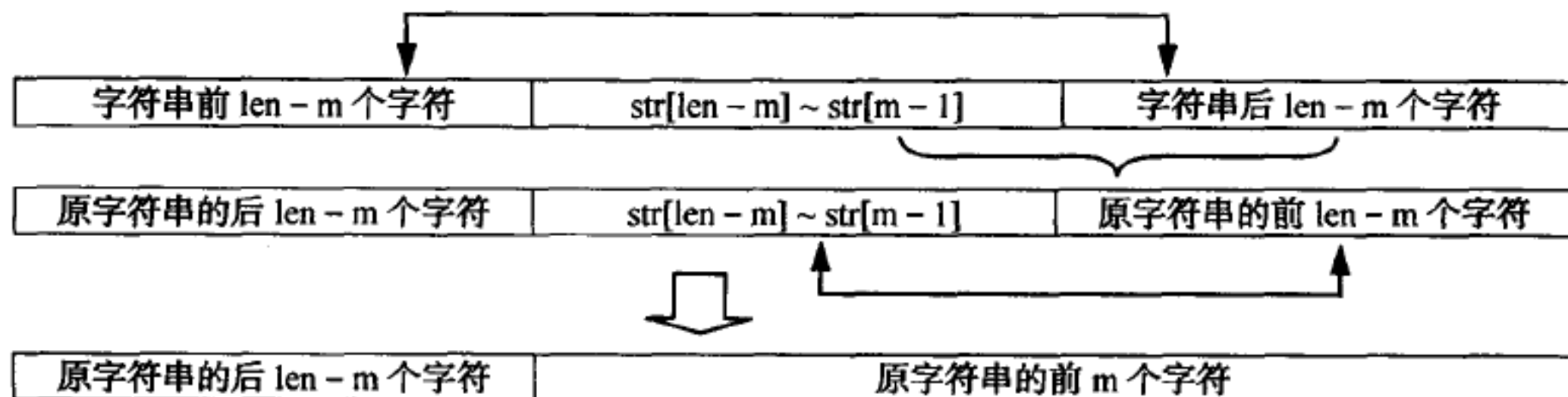
* 4. 示例

[例6.13] 将字符串 str (设长度为 len) 的最前 m 个字符调换到最后 m 个位置。

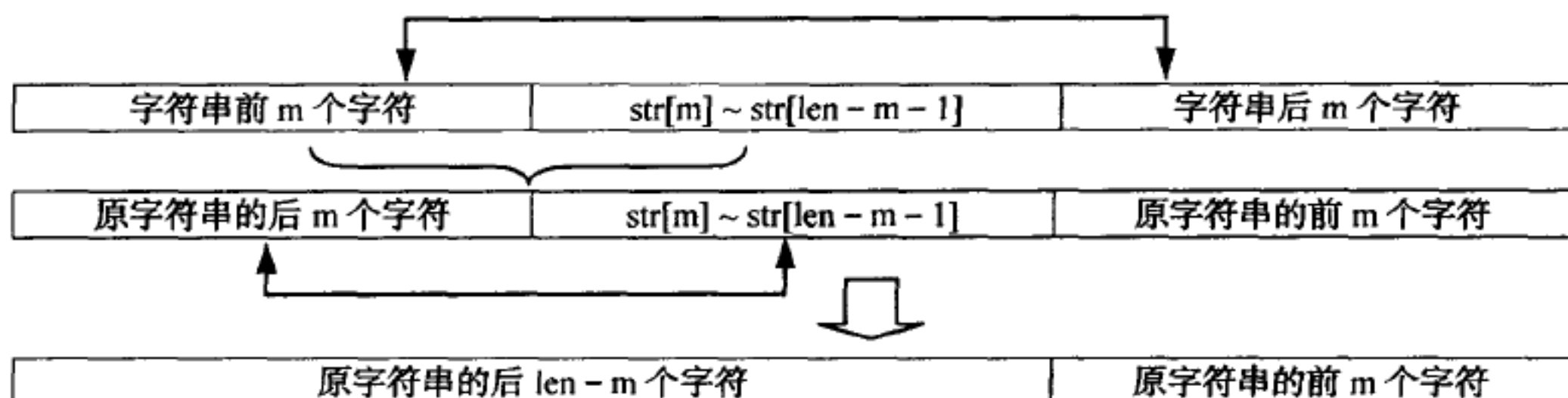
分析: ① 当 $m = \text{len} - m$ 时, 只需将字符串前 m 个字符与后 $m (= \text{len} - m)$ 个字符调换即可。



② 当 $m > \text{len} - m$ 时, 将字符串前 $\text{len} - m$ 个字符与后 $\text{len} - m$ 个字符调换, 剩下的将是调换后字符串 $\text{str}[\text{len} - m] \sim \text{str}[m - 1]$ 与 $\text{str}[m] \sim \text{str}[\text{len} - 1]$ 之间的调换问题。



③ 当 $m < len - m$ 时, 将字符串前 m 个字符与后 m 个字符调换, 剩下的将是调换后字符串 $str[0] \sim str[m-1]$ 与 $str[m] \sim str[len-m-1]$ 之间的调换问题。



可见这是一个递归问题, 下面用函数 `change()` 实现其算法; 函数 `swap()` 用于交换两个等长的字符串内容。

```
#include "stdio.h"
#include "string.h"
void swap(char *s1, char *s2, int len)
{
    char t;
    while(len--)
    {
        t=s1[len];
        s1[len]=s2[len];
        s2[len]=t;
    }
}
void change(char *s, int len, int m)
{
    int n=len-m;
    if(m==n) swap(s, s+m, m);
    if(m>n)
    {
        swap(s, s+m, n);
        change(s+n, m, m-n);
    }
    if(m<n)
    {
        swap(s, s+n, m);
        change(s, n, m);
    }
}
main()
```



```

{
    char str[80];
    int m, len;
    printf("\n 请输入一个字符串:");
    gets(str);
    len = strlen(str);
    printf("\n 请输入要调换的字符个数:");
    scanf("%d", &m);
    change(str, len, m);
    printf("调换后的字符串为: %s\n", str);
}

```

运行结果:

请输入一个字符串:0123456789

请输入要调换的字符个数:3

调换后的字符串为:3456789012

[例6.14] 编写求两个正整数的最大公约数的函数 gcm()。

分析:若正整数 g 是正整数 m 和 n 的公约数,即 g 同时整除 m 和 n ,则:

① g 可以整除 $\min(m, n)$ 和 $|m - n|$;

② 若 m 等于 n 时它们的最大公约数就是它们本身 m (或 n)。

以 $\text{gcm}(m, n)$ 表示正整数 m 和 n 的最大公约数,则求解 $\text{gcm}(m, n)$ 是一个递归问题:

递推关系: $\text{gcm}(m, n) = \text{gcm}(|m - n|, \min(m, n))$, 当 m 与 n 不等时

极端情形: $\text{gcm}(m, n) = m$, 当 m 与 n 相等时

上式递推关系的设计除了满足最大公约数的性质外,还基于算法的有效性,据此设计的递归函数在算法上是确保在有限步达到极端情形,从而得到 m 和 n 的最大公约数。

```

unsigned gcm(unsigned m, unsigned n)
{
    if(m == n) return (m);          /* 递推出口 */
    if(m > n) return (gcm(m - n, n)); /* 递推 */
    else return (gcm(n - m, m));     /* 过程 */
}

```

以上递归函数仅有两条语句,递推出口对应于该递归问题的极端情形;递推过程对应于递推关系。使用循环语句来实现该算法也很简单,但算法的效率不如递归算法。

有些情形使用循环等基本控制结构构造问题的求解过程是一件十分吃力的事情,而使用递归函数则十分自如,如汉诺塔问题。

[例6.15] 汉诺(Hanoi)塔问题:有三根柱子编号为1、2、3,其中1号柱上有若干个大小不等的盘子,大的在下,小的在上(图6.4)。要求将柱1上的盘子全部移到柱3上,在移动的过程中可以借助于2号柱,但一次只能移动一个盘子,且在移动过程中三根柱子上的盘子始终保持大盘在下、小盘在上的秩序。编写完成此项任务的函数,并输出其移动步骤。

分析:将柱 x 上的 n 个盘子借助于柱 y 的帮助(如果需要的话)移动到柱 z 这样一个一般的汉诺塔问题以 $\text{hanoi}(n, x, y, z)$ 表示, x, y, z 各自表示1、2、3三根柱子中的一根。该问题通过

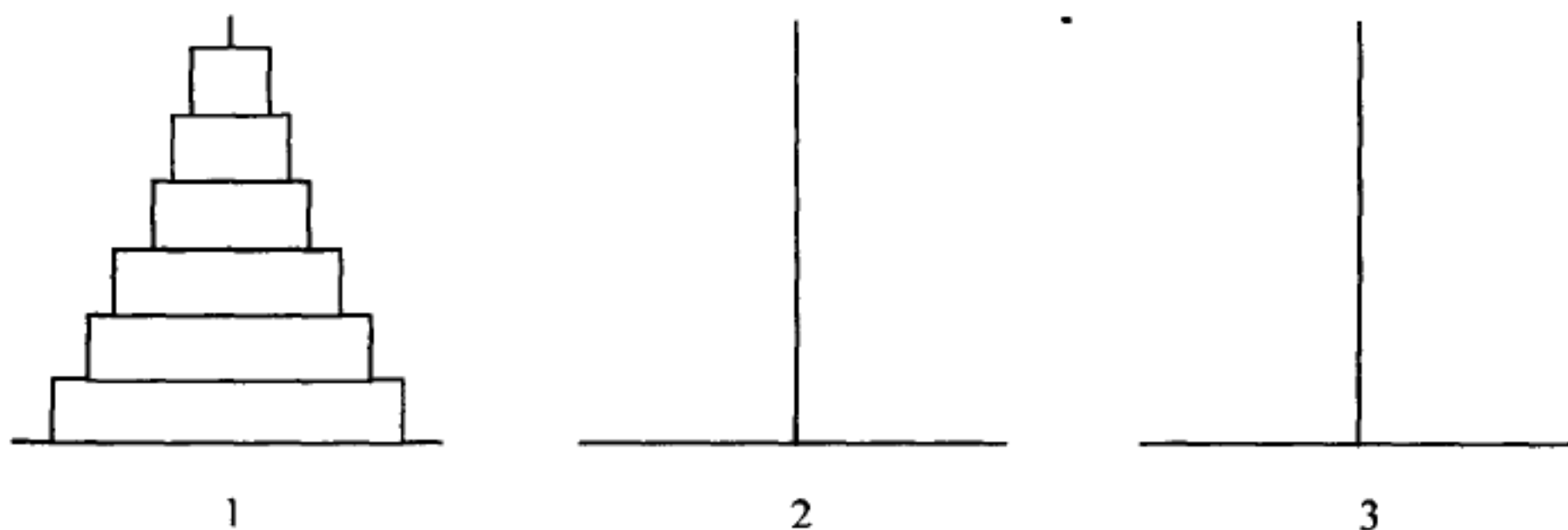


图6.4 汉诺(Hanoi)塔问题图示

循环结构来构造求解是非常困难的,下面我们把它当作递归问题来进行分析。

极端情形:当 $n=1$ 时, $\text{hanoi}(1, x, y, z)$ 问题非常简单——直接将柱 x 上的这个盘子移动到柱 z 上并输出这个移动过程即可结束这个问题。

递推关系:当 $n>1$ 时,可以将 $\text{hanoi}(n, x, y, z)$ 问题分解成以下三个同类的较小规模问题而顺序解决,从而形成递推关系:

- ① $\text{hanoi}(n-1, x, z, y)$: 将柱 x 上面的 $n-1$ 个小盘子借助于柱 z 移动到柱 y 上。
- ② $\text{hanoi}(1, x, y, z)$: 将柱 x 上剩下的那个大盘子直接移动到柱 z 上。输出移动过程。
- ③ $\text{hanoi}(n-1, y, x, z)$: 将柱 y 上的 $n-1$ 个小盘子借助于柱 x 移动到柱 z 上。

以上递推关系可以有效地利用递归函数加以实现,其有效性不仅表现在它满足题设要求:移动过程中大盘子不会出现在小盘子上面,移动过程的输出(代表实际的移动)只发生在 n 等于1的情形;还在于递推过程能在有限步内完成,因为我们在递推关系中对问题的分解始终是按往小规模的方向进行的,有限步后这些分解的问题一定会全部达到极端情形。

汉诺塔问题是一个著名的递归问题,该问题本身简单易懂,但其有效的解法却只有递归解法这一种!下面就是解该问题的递归函数定义。

```
void hanoi(int n, int x, int y, int z)
{
    if (n == 1)
        printf("将一个盘从%d 柱移动到%d 柱\n", x, z);
    else
    {
        hanoi(n-1, x, z, y);
        printf("将一个盘从%d 柱移动到%d 柱\n", x, z);
        hanoi(n-1, y, x, z);
    }
}
```

函数调用 $\text{hanoi}(64, 1, 2, 3)$ 将输出将柱1上的64个盘子按规定移到柱3的整个移动过程。

函数的递归调用是C语言的一大特点,它丰富了C语言的编程,同C语言的许多其他特点相似,合理的使用对于程序设计是锦上添花,而对它们的滥用则是画蛇添足,如[例6.12]的递归函数 $\text{sum}()$ 并没有给我们带来任何好处,反而增加了因函数调用而带来的附加的运行时间和内存空间上的开销。太多的函数调用会影响程序执行的效率,因此在程序设计时就需要在算法的效率和程序结构模块化的简洁清晰间进行取舍权衡,初学者应主要注意使程序的结构

模块合理清楚,这样更有利于程序的正确设计、调试和执行,而不要过于追求程序运行时的高效,其实真正的高效是建立在合理的结构之上的。

6.4 局部变量与全局变量

随着对函数的深入了解,一些关于变量的基本问题浮现出来,即一个变量除了其数据类型外还有一些性质是我们尚未了解的,例如当我们在一个程序文件中写了几个函数时,在其中一个函数中定义的变量能不能在另一个函数中使用?能不能在不同的函数体中定义同名的变量?在两个不同的函数中定义的同名变量之间有没有关系?能不能在函数体之外定义变量?这种变量如何使用?它与函数内定义的变量有什么关系和有什么不同?等等。核心的问题是关于变量起作用的位置以及范围,换句话说就是指对于一个变量,可以在哪里引用它来进行运算和操作?在多大的范围内该变量是有效的?这就导致了局部变量和全局变量概念的引入。

局部变量:在一个函数内部定义的变量称为局部变量,它只在定义它的函数内有效。

全局变量:在所有函数外部定义的变量称为全局变量,它在定义它的文件范围内有效。

可以看出,局部变量与全局变量的定义位置不同,这导致了它们可以有效存取范围不同,遵循以下基本规则:

1. 作用域:变量名只能在定义或说明该变量的程序块(可能是程序文件、函数体或复合语句)内从定义位置到该程序块结束位置范围内进行有效引用,而在该程序块外的引用是非法的。每个程序块内可以各自(平行独立地)定义变量,它们在各自的程序块中起作用。

2. 可见性:一般情况变量通过变量名在其作用域范围内都是可见的,可以通过变量名对其进行存取。但在程序块嵌套中(如程序文件包含函数体、函数体包含复合语句、复合语句嵌套复合语句等),外层程序块中定义的变量,在内层程序块中没有重新定义(即同名变量)时,则在内层继续可见;如果内层一旦定义了同名变量,则相应外层变量在内层被隐藏起来而内层的同名变量起作用,直到内层程序块结束。

变量名的作用域决定变量的可视范围,绝大部分情况下,两者是一致的,在不引起误会的前提下,以后将不区分变量和变量名这两个名词,也不区分作用域和可见性。

在复合语句(分程序)中定义的变量,它被包含在函数中,所以是局部变量,只在函数体内更小的局部范围起作用。在这里可以看到一个变量具有程序块的作用域,根据程序块的不同,可以分为文件域、函数域及语句域(复合语句)。通过以上规则就得到了前面问题的答案:

- 一个局部变量仅在定义它的函数内有效,在其他函数中没有被定义且不可见。
- 不同函数体中或某函数体内平行位置的复合语句中可以定义同名局部变量,它们是不同的变量,没有任何关系,不会互相干扰。
- 可以在函数中嵌套的各复合语句中定义同名局部变量。外层变量仅在外层起作用,而在内层程序块中,内层变量起作用。外层变量不会影响内层变量的取值,内层变量也不会改变外层变量的大小,它们是不同的变量。
- 可以定义全局变量以在整个程序文件范围(包括所有函数体)内对它进行存取。
- 当一个全局变量与某函数体或复合语句中定义的局部变量同名时,在该函数体或复合语句内局部变量起作用,而在之外则全局变量起作用。

另外,函数的形式参数也是局部变量,它与在该函数的函数体中说明部分里定义的局部变

量的作用域是完全相同的。

局部变量的定义位置都是在某函数内部,因此也可称之为内部变量。相反,定义全局变量的位置是在所有函数的外部,所以也称全局变量为外部变量。其实局部变量和全局变量是针对变量的作用域而言的,而内部变量和外部变量则是相对于变量定义所处位置是在函数的内部还是外面而论的,可以认为它们是同一问题从不同角度来看的差异。

[例6.16] 关于全局变量和局部变量的作用域例子。

```
#include "stdio.h"
char *a="全局变量 a", *b="全局变量 b";
main()
{
    void x();
    char *a;
    a="局部变量 a(0)";
    x();
    printf("在函数 main()中可视变量为:%s 和 %s\n",a,b);
}
void x()
{
    char *a,*c;
    a="局部变量 a(1)";
    c="局部变量 c(1)";
    {
        char *c;
        c="局部变量 c(2)";
        printf("在复合语句中可视变量为:%s,%s 和 %s\n",a,b,c);
    }
    printf("在函数 x()中可视变量为:%s,%s 和 %s\n",a,b,c);
}
```

运行结果:

在复合语句中可视变量为:局部变量 a(1)、全局变量 b 和局部变量 c(2)

在函数 x()中可视变量为:局部变量 a(1)、全局变量 b 和局部变量 c(1)

在函数 main()中可视变量为:局部变量 a(0)和全局变量 b

上例中一共定义了四次变量 a,分析程序可知它们是完全不同的变量,另外对于全局变量 b,它是全程序有效的。而全局变量 a 在函数 main()和 x()中都分别被同名的局部变量所“屏蔽”掉了,它在整个程序中没起到任何作用。比较同名局部变量 c,它们也是不同的变量,作用域各自不同。

全局变量与局部变量的作用域不同,可以利用这点将它们用于不同的场合。全局变量具有较大的有效范围,这可以方便地将多个函数公共的参数集中起来设为全局变量而减少函数的参数,或者直接利用全局变量为函数传递原始数据,更多的情况是利用全局变量来加强程序各模块(函数)之间的联系。局部变量被严格地封装在函数的内部而为此函数所“私有”,不会受

外界因素的干扰,有利于函数的独立和通用。在用户自定义函数的编写时应慎用全局变量,一个明显的例子是若两个函数都在运行时改变了同一个全局变量,那么整个程序的运行结果将会因这两个函数的调用次序不同而变化,即使这两个函数原本并没有调用次序的限制,这将使得在编写这两个函数时,除了应考虑函数的功能外,还需注意它们运行时的执行次序以及其中一个的调用对另一个执行的影响,这样模糊了各程序模块之间的界限,增加了程序开发的难度、降低了程序的清晰性;同时全局变量始终占用内存资源,太多或太大的全局变量大量消耗了程序运行时的可利用动态内存空间。

6.5 变量的存储类型与变量的初始化

一个变量对应着计算机中内存存储器的一个或多个存储单元,如在 PC DOS 系统中整型变量占两个字节,按整型格式存取,实型变量占四个字节而按浮点格式存取等。变量的作用域问题则是由该变量的存储方式决定的,具体而言,存储方式决定局部变量以临时变量的方式存储,当定义它的函数被调用或分程序(复合语句)被执行时,它被赋予一块内存空间而存在,可以被存取及引用,而一旦离开了相应的程序块,该内存块被释放而相应局部变量自动消失,因此我们也称这种局部变量为自动变量。全局变量则是在整个程序运行时间内一直是存在的,始终占用存储空间,因而也可以被任何时间正在执行的程序块所存取。

变量的不同存储方式确定了变量的不同存储类型,变量的数据类型与存储类型反映着变量的两个性质。与数据类型一样,变量也有相应的类型修饰符,这些类型修饰符与数据类型说明符一起定义或说明变量。

1. 自动类型修饰符

自动类型修饰符 `auto`,用于定义自动变量,如:

```
auto int a;
```

该定义指出变量 `a` 是一个整型的自动变量,其中关键字 `auto` 可以省略。

自动变量仅用于局部变量一种情形,它的作用域被限制在该变量定义的函数内,自动变量的存在时间也仅是该函数被调用执行的那段时间,这两点性质导致自动变量是最常用的变量类型:自动变量存在于函数内部,使函数模块的编写更自由也更安全;自动变量伴随函数调用的局部存在性,使得在不同函数中可以定义更多及更大的自动变量,它们占用内存的大小不是全部的相加,而仅仅是被调用函数所定义的变量占用内存的大小。

2. 外部类型修饰符

外部类型修饰符 `extern`,用于说明外部变量,如:

```
extern float x;
```

该说明指出变量 `x` 是一个实型的外部变量。其作用域是在程序文件该变量的定义位置到文件结束,那么,若要在该变量定义位置之前使用它,就必须使用关键字 `extern` 说明,通知编译器该变量是一个外部变量;或者在一个多程序文件的应用程序中,一个文件中的程序要使用在另外的程序文件中定义的外部变量,也需要在使用它之前如上说明它是外部的,以便编译器在编译这个程序文件时知道它是一个已定义的外部变量。

自动变量没有变量说明的问题,它的作用域受到严格的限制而不可以其他地方使用。外部变量的定义不需要类型修饰符,其定义位置是在所有函数的外部就确定了它的存储类型。另

外,一个外部变量的变量名应该是在整个程序(可能包含多个程序文件)范围内惟一的,否则编译将报告变量重复定义的错误。

[例6.17] 自动与外部类型修饰符示例。

```
#include "stdio.h"
main()
{
    auto int m=0;
    double x=1.001;
    extern int times;
    settimes();
    for(;m<times;m++) x*=x;
    printf("x=%f",x);
}
int times;
settimes()
{
    times=10;
}
```

在上例中,m 和 x 都是自动变量,关键字 auto 可以省略,times 是外部变量,在主函数中使用前被说明。此说明的有效范围限制在主函数之内,为了获得外部变量说明的更大的有效空间,可以将该说明移到主函数上面的文件头位置上,那么在其后的多个函数(如果有的话)都可以直接引用该外部变量进行运算及操作。

适当地使用外部变量可以为程序设计带来方便,但过量过滥地使用它则不可避免地要引起各种副作用;程序设计时必须明确每一个全局变量的每一次改变对每一个模块的影响并要正确把握住它;程序运行时并不是一直在使用每一个外部变量,但每一个外部变量都始终占用内存资源而可能造成存储空间的紧张。因此我们使用外部变量时应小心谨慎。

外部变量在程序运行时始终占用内存而一直存在并被多个模块所共有;自动变量则是私有的且随着函数调用和返回而产生和消亡。出于程序设计的需要,也是外部变量和自动变量的自然补充,变量可以是一种新的类型:它是函数所私有的,具有局部的可见性,但在程序运行时占用内存,这样它的值可以在每次函数调用时保存下来,这就是静态局部变量。它的定义是在局部变量的定义时加上静态类型修饰符 static。

3. 静态类型修饰符

静态类型修饰符 static,用于修饰静态变量,如:

```
static int m;
```

当使用 static 修饰局部变量时,该局部变量就称为静态局部变量,该静态变量是函数私有的,因而只能在函数内部定义,具有局部变量的作用域性质,但它是静态存储的。事实上,局部变量有两种:静态局部变量和动态局部变量,前面所讨论的 auto 类型自动变量即动态局部变量,在程序运行时函数被调用过程中动态存在,函数返回后消失;而用 static 修饰的局部变量是静态存储的,在整个程序运行过程中静态存在,但它仅在定义它的函数体内可见,因而只能在函数被调用执行时引用它,这样使得当该函数被多次调用时,某次函数调用时可以获得上次该函数

调用完成时该变量的值。

[例6.18] 打印1至 $n!$ 的数列。

```
#include "stdio.h"
void factorial()
{
    static long n=1;
    static int m=1;
    printf("第%d 次调用——— %d 的阶乘为:%d\n",m,m,n);
    m++;
    n *= m;
}
main()
{
    int k=1;
    while(k++ <= 10) factorial();
}
```

上例运行将输出1到10的阶乘。注意在函数 factorial() 中定义了两个静态局部变量 m 和 n ，在此函数调用过程中变量 m 将记录该函数被调用的次数，而 n 则保存 $m!$ 的值。该程序的输出序列应为：

```
第1次调用———1的阶乘为:1
第2次调用———2的阶乘为:2
第3次调用———3的阶乘为:6
第4次调用———4的阶乘为:24
...
```

可见在函数 factorial() 被第一次调用时， m 和 n 均为初值1，调用完成时， m 和 n 的值分别为2和2；函数第二次被调用时 m 和 n 的值为第一次调用完成时的值2和2，它们被保存下来，同样第二次调用完成时 m 和 n 的值被修改为3和6，它们被保存下来为第三次函数调用所用……静态变量 m 和 n 区别于自动变量在于：静态变量不因函数调用完成而消失，其值在不同次数的函数调用时可能是不一样的。但静态变量具有与自动变量一样的局部变量作用域，仅能在定义它的函数体中引用它们。对比不使用静态变量的函数：如在函数 factorial() 中 m 和 n 均定义为自动变量，那么每次该函数调用的结果应是完全一样的：

```
第1次调用———1的阶乘为:1
第1次调用———1的阶乘为:1
...
```

函数中语句 $m++$ ；和 $n *= m$ ；将毫无意义，因为每次函数调用完成后，变量 m 和 n 将消失！

静态变量为程序提供了一种“永久性”存储的手段，静态类型修饰符 static 指出其修饰的变量就像全局变量一样，在程序运行过程中是存储在静态数据区的，因而具有整个程序运行过程的生存期，相对于自动变量的生存期仅在于定义它的函数被调用执行的过程这段时间范围，静态存储局部变量丰富了变量的类型；但静态变量在程序设计中并不是必须的语言成分，且它一直占用内存空间而会造成存储空间的浪费，因此应尽量避免使用它。

静态类型修饰符 `static` 也可用于修饰全局变量。这里的 `static` 并不是用于指明该变量具有“永久”的生存期且存放在静态数据区,因为在前面的讨论中我们已经了解到全局变量的全局本质就决定了这点,相反地,该修饰符是用于指出其修饰的全局变量是“本地”的!当定义全局变量时冠以静态类型修饰符 `static` 时,该全局变量被定义为静态全局变量,它的有效范围仅在该程序文件范围之内,此时它对于其他的程序文件中是被“隐藏”的,因而不能对它进行引用,就像局部变量仅在定义它的函数里有意义一样,例如:

程序文件 file1.c

```
static int m;
main()
{...}
myfun1() {...}
myfun2() {...}
```

程序文件 file2.c

```
static float m;
myfun3() {...}
myfun4() {...}
```

在程序文件 file1.c 中定义外部整型变量 `m` 为静态变量,一方面使它具有全局特性,能为函数 `main()` 及 `myfun1()`、`myfun2()` 所用,另一方面使得该变量的可视范围仅局限于本程序文件而对其他的程序文件隐藏,避免可能的重名变量定义之类的冲突,这样我们就可以在另一个程序文件 file2.c 中定义静态的外部浮点变量 `m`,两个变量毫无关系,减少了程序文件之间的数据干扰。

4. 寄存器类型修饰符

一般变量都保存在内存中,C 语言也允许所定义的变量保存在寄存器中,称之为寄存器变量,因此,寄存器类型修饰符 `register`,用于定义寄存器变量,如:

```
register int k;
```

寄存器是在中央处理单元(CPU)中的存储器,它的数目极少但存取速度极快(比对内存存储器的存取速度快很多)。寄存器变量常用于需频繁存取操作的变量,如循环变量,如:

[例6.19] 寄存器变量示例。

```
main()
{
    register int m,sum=0;
    for(m=1;m<100;m++) sum += m;
    printf("1至100的和等于:%d",sum);
}
```

一般对寄存器变量的使用都有限制(因系统的不同而有差别),若定义了过多的寄存器变量,则可能仅前几个保存在寄存器中,而其他的作为自动变量放在内存中。

变量有四种存储类型:`auto`、`extern`、`static` 和 `register`。从变量存储的物理器件上看,有自动变量和寄存器变量;从变量在程序运行时的生存期(时间角度)来看,有动态变量(自动变量)和静态变量;从变量的定义位置或其作用域(空间角度)来看,有内部变量或局部变量和外部变量或全局变量。

5. 变量初始化

变量初始化是指定义变量的同时赋以初值,在只有变量定义而没赋以初值的情况下,按照规定,外部变量和静态变量的初值总是被赋予零(如整型变量被赋值0,字符型变量被赋值'\0'及浮点型变量被赋值0.0等等),而自动变量和寄存器变量的值是不确定的。数组的情形也类

似。

简单变量(非数组或结构)在它们被定义时,可以通过跟在变量名后面的赋值号和常数表达式来对其进行初始化,如:

```
float x=1.0;
int length='Z'-'A'+1;
```

上述初始化工作与变量先定义,后赋值是等价的:

```
float x;
int length;
x=1.0;
length='Z'-'A'+1;
```

对于外部变量和静态变量的初始化工作是在编译时一次性完成的,该变量在程序运行之初就具有了初值,第一次对它们的赋值操作就会使原初值不复存在;自动变量和寄存器变量的初始化工作则是在程序运行时每次函数调用都要进行的,因而每次函数调用开始时,它们都等于显式给出的初值。例如:

```

      实参1    实参2
      ↓      ↓
int rect(int width,int height)
{
    int leftx=0, lefty=0;
    int rightx=leftx+width, righty=lefty+height;
    ...
}
```

6.6 外部函数与内部函数

前面讨论了变量的有效范围问题,对于函数,它的有效范围又是如何的呢?即我们可以在哪里有效地调用它?这就是本节要讨论的问题。

函数在本质上是外部的,函数名同变量名一样遵循作用域规则。C 语言不允许函数的嵌套定义,各个函数之间是平行并列的关系,互相之间可以调用,因而函数具有全局的有效范围。但每个函数定义都附属于某个程序文件,类似于外部变量,我们可以决定该函数的可视范围是所有程序文件还是仅仅限于该函数定义所处的文件,从而对应着外部函数和内部函数。

1. 外部函数

在定义函数时,如果冠以关键字 `extern`,表示该函数是一个外部函数,如:

```
extern void myfun() {...}
```

这样定义的函数 `myfun()` 的可视范围是所有的程序文件,即它可以被应用程序中的任何其他函数所调用而不论这个函数是否与函数 `myfun()` 所处同一个程序文件。如果定义函数时省略关键字 `extern`,则隐含为外部函数,前面学习中所遇到情况均如此。

与外部变量一样,当在其他程序文件中调用此函数之前,应该先用关键字 `extern` 说明该函数是一个外部函数,此说明可以放在调用它的函数体说明部分,也可以是在调用它的函数定义前面。

2. 内部函数

在定义函数时,如果冠以关键字 `static`,表示该函数为一个内部函数,如:

```
static void myfun() {...}
```

这样定义的函数 `myfun()` 的可视范围是定义它的程序文件,即该函数被限制为仅能被本程序文件中的函数所调用,如果在不同的文件中有同名的内部函数,它们互不干扰,这样可以方便大型应用程序的编写工作。通常仅由某程序员负责的并且互相联系的外部变量及函数放在一个文件中,并用关键字 `static` 来定义使之本地化,这个文件也就相应地独立了,减少了程序出错的机会。

* 3. 示例

关于函数的可视范围其内容是简单和容易理解的,下面程序包括这一点内容,也包括了外部变量可视范围的内容及各种存储类型的变量应用的内容,以作类比和小结。但更多的是关于程序设计的问题,如各种控制结构的典型应用,堆栈这种非常有用的数据结构的实现和操作等等,可以作为阅读练习,以提高程序设计的能力。

[例6.19] 计算器程序(即算术表达式求值程序)。

程序文件 `file1.c`

```
#include "stdio.h"
#include "stdlib.h"          /* 包含系统函数 atof() 的原型 */
#define NUM '0'              /* 数字标记 */
extern char getop(int *);
extern void push(double);
extern double pop(void);
extern void init(void);
char s[100];                 /* 用以记录算术表达式(后缀表达式表示) */
static char getop(int * pointer)
{
    static int p=0;
    * pointer=p;
    if(s[p]>='0' && s[p]<='9' || s[p]=='.')
    {
        while(s[p++]!='\0');
        return ('0');
    }
    else return (s[p++]);
}
main()                       /* 计算器程序 */
{
    char type;                /* 类型标记 */
    double op2;               /* 双目操作的第二个操作数 */
    int place;                /* 指示 s 表达式已处理的字符长度 */
    init();
```



```

while((type=getop(&place))!='\n')
{
    switch(type)
    {
        case NUM : push(atof(s+place)); break;
        case '+' : push(pop()+pop()); break;
        case '*' : push(pop()*pop()); break;
        case '-' : op2=pop(); push(pop()-op2); break;
        case '/' : op2=pop();
                    if (op2==0)
                        printf("错误:除零错!\n");
                    else
                        push(pop()/op2);
                    break;
        case '#' : printf("其结果是:%f\n",pop()); break;
        default: printf("后缀表达式错误\n"); break;
    }
}
}

```

程序文件 file2.c

```

#include "stdio.h"
#define MAXNUM 100 /* 运算符栈的深度 */
extern char s[];
static char opr[MAXNUM]={'#'}; /* 运算符栈, # 为虚设的运算符 */
static int sp=1; /* 运算符栈顶指针 */
static void push(char c) /* 进栈操作 */
{
    if(sp<MAXNUM) opr[sp++] = c;
    else printf("出错:栈已满\n");
}
static char pop(void) /* 出栈操作 */
{
    if(sp>0) return (opr[--sp]);
    else printf("出错:栈已空\n"); return ('\0');
}
static int prior(char c) /* 计算操作符优先级 */
{
    switch(c)
    {
        case '#': return (0); /* 虚设的运算符 # 的优先级最低 */

```

```

        case '(': return (1);
        case '+': case '-': return (2);
        case '*': case '/': return (3);
        default: return (4);          /* 不对非法运算符进行处理 */
    }
}

void init()          /* 输入算术表达式并转换为后缀表达式 */
{
    char x,y;
    int m=0;
    printf("请输入一个合法的算术表达式:");
    while((x=getchar())!='\n')
    {
        if(x==' ') continue;        /* 除去算术表达式中的所有空格 */
        if(x=='(')                  /* 处理左括弧——进栈,等待右括弧出现 */
        {
            push(x);
            continue;
        }
        if(x>='0' && x<='9' || x=='.' ) /* 处理数字 */
        {
            for(s[m++]=x;(x=getchar())>='0' && x<='9' || x=='.';s[m++]=x);
            s[m++]='\0';              /* 以空字符作为一个数字的结束标志 */
        }
        if(x=='\n') break;
        if(x==')')                  /* 处理并除去原算术表达式中的括弧 */
            for(y=pop();y!='(';y=pop()) s[m++]=y; /* 输出括号内的运算符 */
        else                        /* 处理其他算术运算符 */
        { /* 将栈内较高及相同级别(应先计算)的运算符输出到后缀表达式 */
            for(y=pop();prior(x)<=prior(y);y=pop()) s[m++]=y;
            push(y);                /* 将最后取出的较低级别的运算符压栈 */
            push(x);                /* 将当前运算符压栈 */
        }
    }
    while(sp>0) s[m++]=pop(); /* 将栈中剩余运算符输出到后缀表达式 */
    s[m]='\n';                  /* 以回车符作为后缀表达式的结束标志 */
}

```

程序文件 file3.c

```
#include "stdio.h"
```

```
#define MAXNUM 100
```

```
/* 数据栈深度 */
```

```

static double val[MAXNUM];      /* 操作数栈 */
static int sp=0;                /* 操作数栈顶指针 */
extern void push(double v)      /* 进栈操作 */
{
    if(sp<MAXNUM) val[sp++] = v;
    else printf("出错:栈已满\n");
}
extern double pop(void)          /* 出栈操作 */
{
    if(sp>0) return (val[--sp]);
    else printf("出错:栈已空\n");
    return (0.0);
}

```

运行结果:

```

1+2*((3+4)*5.6-7.8)/9 /* 键盘输入 */
7.977778

```

上例程序有一定的复杂性,撇开其算法不管(若有兴趣可查阅数据结构方面的书籍,几乎每一本都会讨论堆栈和表达式计算的算法实现),该程序给出了有关变量各种存储类型(register 除外)和外部及内部函数的综合示例:

(1)在 file1.c 中定义了外部的字符数组 s 和说明了外部函数 init()、pop()及 push(),其中外部数组 s 在文件 file1.c 中的主函数 main()、函数 getop()以及文件 file2.c 中函数 init()所使用,外部函数 init()在 file2.c 中定义,外部函数 pop()和 push()在 file3.c 中定义。

(2)在 file2.c 中定义了内部函数 pop()和 push(),它们仅被同文件中的 init()函数所调用,且不与 file3.c 中的同名函数相冲突。文件 file2.c 中的三个函数紧密联系一起完成相对独立的功能。

(3)文件 file3.c 中的两个外部函数 pop()和 push()具有一定的通用性,组织在一起为 file1.c 中的 main()函数所调用。

(4)在 file2.c 和 file3.c 中都定义了一定数目的静态外部变量及数组,它们仅为其所在的程序文件中的多个函数使用。

程序中的几个相关概念:栈是一种数据结构,它具有“先进后出”的操作特征,即若我们将数0,1,2,3,4,...,9依次存入栈中,要想取出数4的话就必须先将其后的数9,8,7,6,5取出,然后才能取出4来。本程序中用两个数组 opr 和 val 来分别实现操作符和数据两个栈结构,函数 pop()和 push()分别用来实现出栈和入栈两个相对的栈操作,变量 sp 是栈顶指针,它的增减分别对应着入栈和出栈两项操作。

算术表达式常规是以中缀表达式的形式出现的,即操作符出现在它的两个操作数中间(本程序只对进行双目运算的表达式进行计算),如 $(1-2)*(3+4)$ 等。后缀表达式则不然,操作符是在相应操作数的后面且与各个运算的先后次序有关, $(1-2)*(3+4)$ 的后缀表达式为1,2,-,3,4,+,*。在后缀表达式下可以依次一次性地进行计算处理:((1,2,-),(3,4,+),*),它已将算术运算的优先级不依赖于括弧和人工判断地表示在后缀表达式的次序之中。更一般的算术表达式 $a+b*(c-d)-e/f$ 的后缀表达式为abcd-*+ef/-,其中a,b,c,d,e,f表示参

与计算的数据值,对该后缀表达式进行数值的算术计算处理时无需再考虑各操作符的算术优先级:((a,(b,(c,d,-),*),+),(e,f,/),-))。

6.7 编译预处理

编译预处理功能是C语言的一大特色,是它与其他高级语言的一个重要区别。C语言允许在程序中使用几个特殊的命令——编译预处理命令,这些特殊的命令在应用程序被正式编译前首先被“预处理”,然后编译系统将这些处理的结果与其他语句一起进行通常意义下的编译,从而得到目标代码。

与一般C语句相区别,C语言规定编译预处理命令均以符号#开头,符号#与其后的命令关键字之间不能有空格。编译预处理功能主要有以下三种:

1. 文件包含: #include

2. 宏定义: #define

#undef

3. 条件编译: #if

#ifdef

#ifndef

#else

#endif

6.7.1 文件包含

文件包含是指在一个文件中,可以把其他文件的内容包含进来,其格式为:

#include <文件名> 或 #include "文件名"

当源程序被编译时,#include命令后用所指定文件名的内容所替换。在进行替换时,前一种形式下编译系统将只在系统指定的系统目录下进行查找该文件;后一种形式下首先在当前目录下进行查找,若找不到,则到系统目录下继续查找。

正如前面学习中所见到的#include "stdio.h",其中stdio.h是C语言提供的标准包含文件(头文件),它包含了一些系统提供的标准输入输出函数的原型说明、外部变量说明及相关宏定义等内容。我们也可以编写自己的头文件,相应包含自己定义的符号常量、外部变量和函数的说明、typedef等内容,然后也利用#include命令放在需要它的位置上,将其内容包含进来,以减少重复劳动和出错的机会。另外,一个包含文件可以包含另外的包含文件,这样使得我们在使用包含文件时更为自由。

6.7.2 宏定义

宏定义有两种形式:不带参数的宏定义和带参数的宏定义。

1. 不带参数的宏定义

#define 标识符 字符串

该定义中,一次只能定义一个标识符,行尾也没有分号(区别于变量的定义)。在这里,标识符也叫宏名。在源文件被编译时,源文件中出现的所有与宏名相同的标识符均用相应宏定义中标识符后的字符串来替换。这种形式常用于定义符号常量,宏名常用有含义的大写字母表示以区别于变量名。

```
#define PI 3.14
```

这里定义了一个宏名 PI,在编译预处理时,程序文件中所有出现标识符 PI 的地方均用字符串 3.14 替换,称宏展开,在真正编译时它才作为常数处理,如同我们原本在该位置书写了 3.14 一样。

[例3.20] 对键盘键入字符进行计数,回车结束。

```
#include "stdio.h"
#define GET getchar()
#define END '\n'
main()
{
    int count=0;
    while(GET != END) count++;
    printf("共键入了%d 个字符\n",count);
}
```

2. 带参数的宏定义

```
#define 宏名(参数表) 字符串
```

如: #define MAX(a,b) a>b?a:b

在编译预处理时,凡宏名 MAX 处将被其后的字符串 a>b?a:b 所替换,而字符串中的参数将由宏名中相应参数值所替换。如宏 MAX(x,3) 将被字符串 x>3?x:3 所替换。

带参数的宏与函数很像,但绝不是一回事,应小心地使用。

[例6.21] 求一个数的平方。

```
#include "stdio.h"
#define SQUARE(x) x * x
int square(int x) { return (x * x); }
main()
{
    int m=10;
    printf("10的平方为:%d=%d\n",square(10),SQUARE(10));
    printf("11的平方为:%d≠%d\n",square(10+1),SQUARE(10+1));
    printf("11的平方为:%d≠%d\n",square(++m),SQUARE(++m));
}
```

运行结果:

10的平方为:100=100

11的平方为:121≠21 /* SQUARE(10+1)的宏展开为10+1*10+1 */

11的平方为:121≠132 /* SQUARE(++m)的宏展开为12*11 */

若将 #define SQUARE(x) x * x 改为 #define SQUARE(x) (x * x),则函数与宏是一致

的,因此,使用宏替换要十分谨慎。

还可以利用已定义过的宏去定义新的宏,如:

```
#define Swap(x,y) { int t; t=x;x=y;y=t; }
#define Order2(a,b) if(a>b) Swap(a,b)
#define Order3(a,b,c) Order2(a,b);Order2(b,c);Order2(a,c)
```

宏名的有效范围是从定义位置到文件的结束,除非我们在文件结束前使用了 `#undef` 命令,该命令取消这个宏名,在该命令之后将不再进行宏替换。其格式如下:

```
#undef 宏名
```

可以用它取消一个标识符的某个宏定义,而对该标识符重新进行其他的定义,但主要还是利用 `#define ... #undef` 来限制宏替换的作用范围,明确在那个程序段中使用宏替换。

6.7.3 条件编译

通常程序的所有行都会被编译,而在条件编译预处理命令下,可以对程序段进行选择编译。条件编译有几种格式:

1. #ifdef

```
#ifdef 标识符
    程序段1
#else
    程序段2
#endif
```

编译预处理命令是对标识符已被 `#define` 定义为宏名时,对程序段1进行编译,否则编译程序段2。

2. #ifndef

```
#ifndef 标识符
    程序段1
#else
    程序段2
#endif
```

此情况正好与1相反:若标识符未被 `#define` 定义为宏名时,对程序段1进行编译,否则编译程序段2。

3. #if

```
#if 常数表达式
    程序段1
#else
    程序段2
#endif
```

当常数表达式(符号常量及常量构成的表达式)的值为真(非零)时编译程序段1,否则编译程序段2。

以上三种情况的 `#else` 及程序段2这部分可以没有,等价程序段2为空语句的情形。各程序

段可以是语句组,也可以是编译预处理命令。

习 题

一、选择题

6.1 C 语言规定,函数返回值的类型由【1】所决定。

- 【1】 A) return 语句中的表达式类型
 B) 调用该函数时的主调函数类型
 C) 调用该函数时的形参类型
 D) 在定义该函数时所指定的函数类型

6.2 下面程序的输出结果为【2】。

```
#include <stdio.h>
#define SQR(x) x * x
main()
{
    int a=10,k=3,m=2;
    a=SQR(k+m);
    printf("%d\n",a);
}
```

- 【2】 A) 25 B) 11 C) 5 D) 10

6.3 下面程序的输出结果为【3】。

```
#include <stdio.h>
main()
{
    int a=10;
    {
        int a=15;
        printf("a. 1=%d",a);
    }
    printf("a. 2=%d\n",a);
}
```

- 【3】 A) a. 1=10,a. 2=10 B) a. 1=15,a. 2=10
 C) a. 1=15,a. 2=15 D) a. 1=10,a. 2=10

6.4 下面程序的输出结果为【4】。

```
#include <stdio.h>
fun()
{
    static c=3;
    c++;
}
```

```

    return(c);
}
main()
{
    int i,k;
    for( i=0; i<2; i++) k=fun();
    printf("%d\n",k);
}

```

【4】 A)3 B)5 C)4 D)6

6.5 下面程序的输出结果为【5】。

```

#include <stdio.h>
void fun( char *s );
main()
{
    static char str[]="123";
    fun(str);
}
void fun( char *s )
{
    if( *s )
    {
        fun( ++s );
        printf( "%s\n", --s );
    }
}

```

【5】 A)3 B)123 C)1 D)3

32	12	12	23
321	1	123	123

二、填空题

6.6 以下程序打印10到99中能被3整除且至少有一位是5的数。

```

main()
{
    int k;
    【1】
    for( k=10; k<=99; k++ ) 【2】 (k);
}
void sub( 【3】 )
{
    int a1, a2;
    a1= 【4】 ;
}

```

```

    a2=m/10;
    if( k%3==0 && (【5】)) printf("%d\n",m);
}

```

三、改错题

6.7 以下程序用来统计字符串 s 中空格的个数,改正其中的错误。

```

#include <stdio.h>
Main()
{
    char str[255];
    gets(str);
    printf("空格个数为:%d\n", fun(str));
}
void fun(char s)
{
    int n=0;
    char *p=&s;
    while(* (p++))
        if(*p==' ') n++;
    return n;
}

```

四、程序设计题

6.8 用库函数 sqrt() 写一个函数,计算一个正实数的四次方根。

6.9 编写一个函数,反序输出一个字符串。

6.10 编写一个函数,其功能是对给定的一个时间数(秒为单位),以“时:分:秒”的格式输出。

6.11 编写函数。计算二次多项式 $f(x) = ax^2 + bx + c$ 的值,对于给定的系数 a, b, c 和 x 的一个区间,以 0.1 为 x 的步长打印 $f(x)$ 的值,找出 $f(x)=0$ 的解及 $f(x)$ 的极大(小)值。

6.12 编写递归函数,计算勒让德多项式:

$$P_n(x) = \begin{cases} 1 & \text{当 } n = 0 \\ x & \text{当 } n = 1 \\ ((2n-1)x - P_{n-1}(x) - (n-1)P_{n-2}(x))/n & \text{当 } n > 1 \end{cases}$$

在递归函数中加入一些输出语句打印 n, x 及中间变量的值,观察递归过程。

第 7 章

数组、指针、函数的应用

前几章中,我们分别系统地描述了有关数组、指针、函数的基本概念及内容,现在此基础上,我们将进一步讲述它们相互结合的应用形式与基本功能。

本章重点讲述在函数间利用数组与指针传递数据的机理与形式。其次,讲授函数与指针相结合的函数指针与指针函数,它们较难以掌握,我们只作一般了解。数组指针与指针数组是指针与数组相结合的形式,它们具有两者的属性,应根据它们的特征,进一步开发其应用。特别是主函数 main 的带参形式在命令行参数中的使用,是一种较典型的用法;还有单向链表,它是指针与结构相结合的一种简单应用,是《数据结构》课程的基础。

本章建议课堂讲授 8 学时,上机 4~8 学时,自学 8 学时。

7.1 概述

1. 在 C 语言程序中,程序不论大小,主要是由函数和变量组成的,函数出现的次数可以是任意的,且是相互独立的。而函数之间或函数与外部变量之间存在着必然的数据联系,这种联系是通过函数间的数据传递建立的,如外部变量或程序中的数据可传递给函数内部;函数内部加工过的数据或函数返回值又可传递给外部变量或外部程序……这种函数间数据通信体现着函数的应用。

2. 函数间的数据传递方式有三种:实、形参结合,函数返回值和外部变量。

3. 数据的传递必须有传递方和接受方。当某些数据要传递时,必须在某函数中通过对另一函数的调用形式来表现,则某函数就是传递方,也称之为主调函数。该函数调用形式中含有能肩负传递某些数据的参数表,称之为实参表,在调用另一函数之前,实参表中的每个参数都必须具有实参值。另一个定义函数就是接受方,也称被调函数,在函数定义中必须有能接受传递过来的参数表,称之形参表,其形参的个数、类型、顺序原则上应与实参表中一致。

4. 被调函数中所有形参及函数体中的变量在主调函数中执行函数调用时,系统才临时给它们分配存储空间,并转向执行被调用的函数,一旦被调函数执行完毕,其相应的所有存储空间被释放,再转回主调函数的调用处。

5. 被调函数执行完毕转向主调函数的调用处,可以带回该函数的返回值,也可以没有,依实际需要而定。

6. 函数调用并不都具有参数,也有一种无参传递的函数调用与函数定义形式,其作用表现在两方面:

(1)被调函数可以独立完成所需功能,与主调函数无关。例如:

fun()


```

{
    int a=3,b=5;
    printf("%d\n",a*b);          /* 被调函数完成输出任务 */
}
main()
{
    fun();
}

```

(2)作为用户应用系统中的一个活动接口,便于今后系统功能的扩充。例如:

```

main()
{
    int i;
    printf("1. 输入 2. 查询 3. 统计 4. 维护 5. 其他 6. 退出\n");
    printf("    请选择:");
    scanf("%d",&i);
    switch(i)
    {
        case 1: fun1(); break;
        case 2: fun2(); break;
        case 3: fun3(); break;
        case 4: fun4(); break;
        case 5: fun5(); break;
        case 6: exit(0);
    }
}
...

```

fun1()、fun2()、fun3()、fun4()均有具体函数定义,以实现对应功能,而 fun5()则可定义为:

```

fun5()
{
}

```

该函数现在什么都不做,一旦将来需要扩充功能时再加以补充。

7. 在函数调用过程中,遵循实参值向形参的单向传递,不容许形参值向实参传递。

8. 传递方式分按数值与地址传递方式,其中以地址传递方式为重点。

7.2 函数之间的数据传递

7.2.1 函数数据按数值传递

按值传递(如图 7.1 所示)中,实参可以是变量,也可是常量,而形参只能是变量。

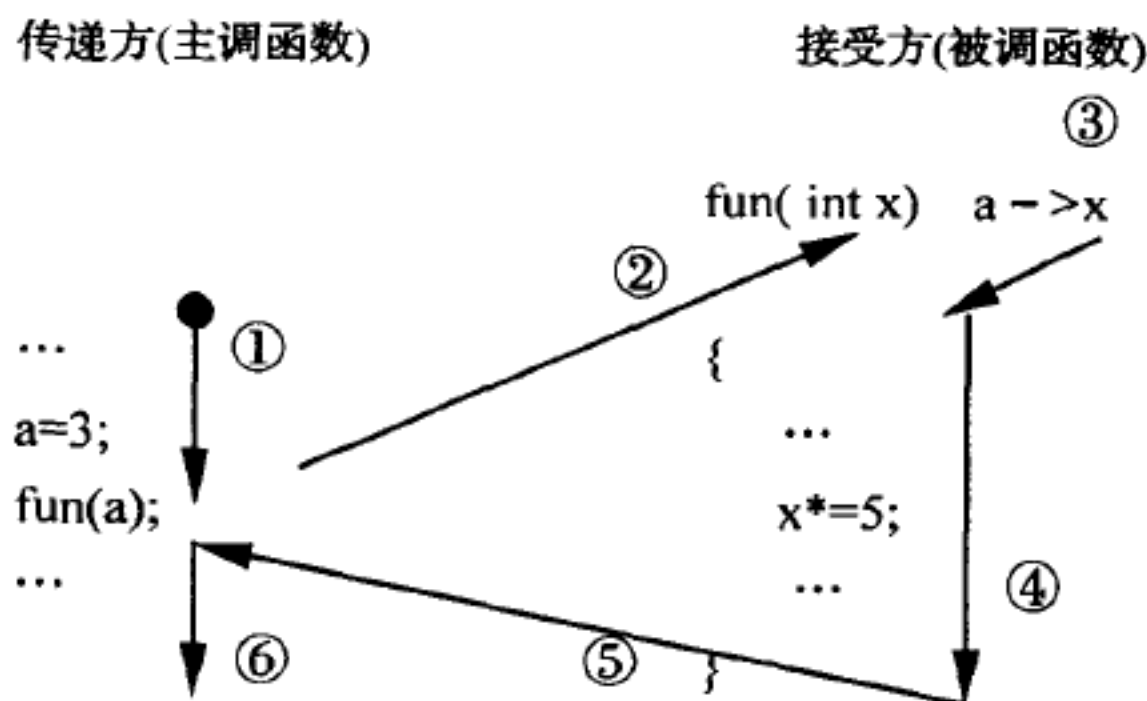


图 7.1 程序调用函数的执行过程

1. 传递过程

主调函数中调用 `fun(a)` 的实现步骤(见图 7.1)如下:

(1) 程序执行完程序段①后,程序会暂时终止顺序执行而转向被调函数 `fun()` 的入口处②;

(2) 系统给形参 `x` 分配存储空间,并将实参 `a` 的值赋给 `x`③,而实参的值保持不变;

(3) 执行 `fun` 函数的函数体④,函数体中定义其他的变量,则此时系统也分配存储空间。在函数体中对数据进行处理后,遇到 `return` 语言或函数中最后的“`}`”,则返回⑤,并释放 `fun` 函数体中变量的存储空间;

(4) 控制流程返回到 `fun()` 函数的调用处,继续执行主调函数剩余的程序段⑥。

2. 示例

[例 7.1] 将一字符常量传递给 `fun` 函数。

```
#include <stdio.h>

fun(char x)
{
    printf("%c\n", x);
}

main()
{
    fun('a')
}
```

运行结果:

a

在上例中,当 main 主函数调用 fun 函数时,先把 a 的值赋给变量 x 后,进入函数体执行打印语句,而后释放 x 变量的存储空间,最后返回主函数。

[例 7.2] 观察一整型数据 5 传递给 fun 函数的前后变化。

```
#include <stdio.h>
fun(int x)
{
    int b=10;
    printf("x_befor=%d",x);
    x*=b;
    printf("x_after=%d ",x);
}
main()
{
    int a=5;
    printf("a_befor=%d ",a);
    fun(a);
    printf("a_after=%d ",a);
}
```

运行结果:

a_befor=5 x_befor=5,x_after=50 a_after=5

3. 小结

(1)函数按值传递是将实参值赋给形参的过程,实参可以是变量,也可是常量,形参只能是变量。

(2)实参与形参所占有空间不是同一存储空间,形参在调用时才分配存储空间,一旦调用完毕,其存储空间就被释放。

(3)当形参接受实参值后,形参在被调函数中可进行任何加工处理,取得新的变量值,但该值不能影响主调函数中的实参值。

7.2.2 函数数据按地址传递

我们还可以用地址作为函数的参数来进行数据传递,而表示地址值的形式有:数组名、指针变量、字符串、& 变量,因此,这些都可作为函数的参数,下面分别加以描述。

1. 以数组名作为函数的实参

数组名是指针常量,在数组定义时,已由系统分配了一组连续的地址空间,其大小为:

sizeof(数据类型)* 数组元素个数

其中首地址值用数组名表示。如图 7.2 所示,至于首地址值究竟是多少,用户完全不必知道,我们所使用的数组名就是首地址。

当用数组名作为函数的实参时,实参传递的值为地址,因而,形参必须为能接受地址值的

变量,那就是数组名与指针变量。

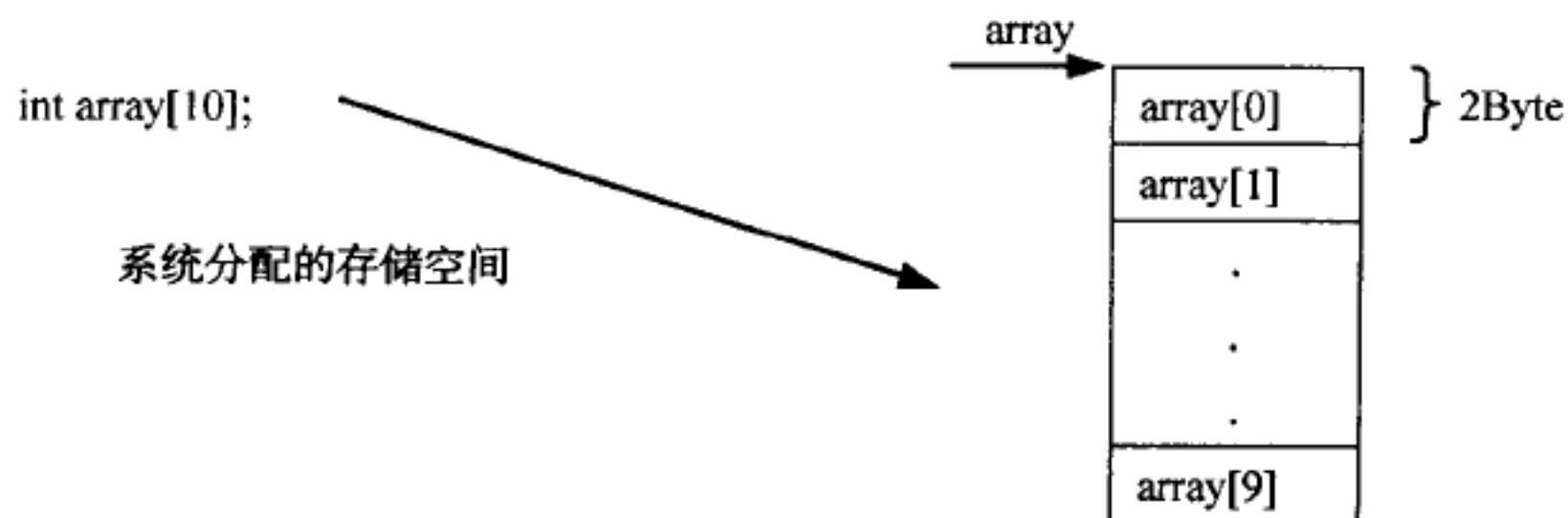


图 7.2 一维数组元素的地址的表示形式

(1) 形参为数组名

当调用函数时,实参数组名的地址值传递给形参数组名,则形参数组名与实参数组名具有同一地址值,即两个数组共享同一存储单元,每个元素既可用实参数组名表示,也可用形参数组名表示(如图 7.3 所示)。

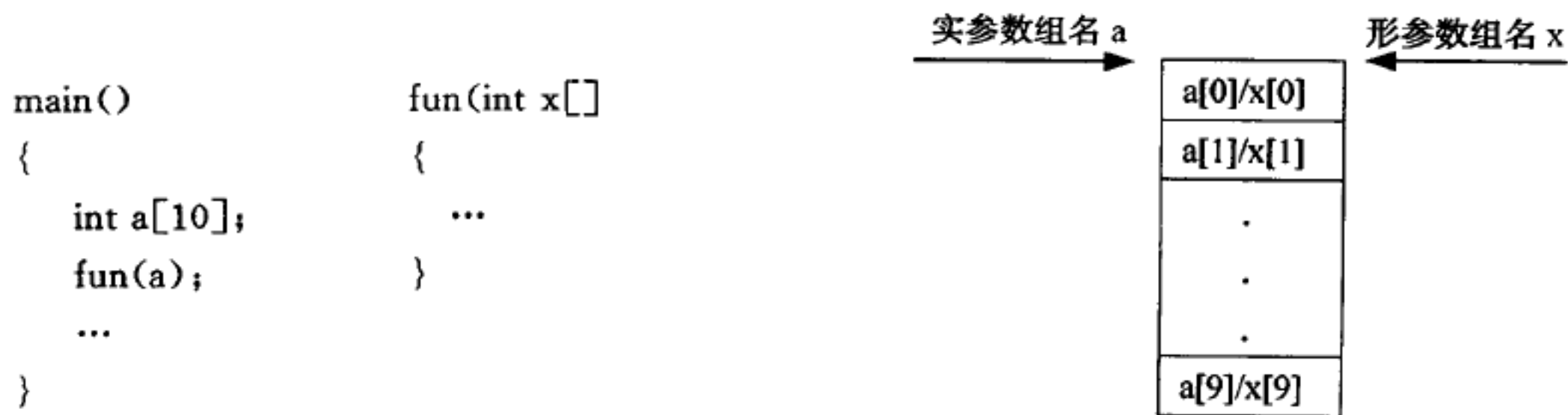


图 7.3 实参、形参都为数组名

因此,在被调函数中改变形参数组元素的值,就是直接改变着主调函数中数组元素的值,但实参传递给形参的地址值并未因此而改变,它仍然遵循单向传递的原则,所以,被调函数中改变形参数组元素值能直接影响实参数组元素值的改变,这一事实并不意味着形参值能反向传递给实参。

[例 7.3] 观察一个一维数组中数组元素值在执行被调函数 fun 前后的变化。

```
#define M 10
#include <stdio.h>
fun(int x[])
{
    int i;
    for(i=0;i<M;i++)
        x[i] *= i;
}
```

```

main()
{
    static int a[]={0,1,2,3,4,5,6,7,8,9},i;
    printf("\n 数组 a 的元素在 fun 函数调用前的值是:  ");
    for(i=0;i<M-1;i++)
        printf(" %2d,  ",a[i]);
    printf(" %2d\n",a[i]);
    printf("fun 函数在调用前数组 a 的地址:  ");
    printf(" %p\n",a);
    fun(a);
    printf("数组 a 的元素在 fun 函数调用后的值是:  ");
    for(i=0;i<M-1;i++)
        printf(" %2d,  ",a[i]);
    printf(" %2d\n",a[i]);
    printf("fun 函数在调用后数组 a 的地址:  ");
    printf(" %p\n",a);
}

```

运行结果:

数组 a 的元素在 fun 函数调用前的值是: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

fun 函数在调用前数组 a 的地址: OEB4:OFD4

数组 a 的元素在 fun 函数调用前的值是: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

fun 函数在调用后数组 a 的地址: OEB4:OFD4

从上例我们可以看到数组 a 的地址没有变化,但数组 a 中的元素却被改变。

[例7.4] 将一个字符串中的字符按上升排序(采用选择法)。

```

#include <string.h>
sort(char x[],int n)
{
    char min;
    int i,j;
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
            if(x[i]>x[j])
            {
                min=x[i];x[i]=x[j];x[j]=min;
            }
}
main()
{
    static char a[]="my computer!";
    int num;

```



```

    num=strlen(a);
    sort(a,num);
    printf(" %d%s\n",n,a);
}

```

运行结果:

```
12 !cemmoprtuy
```

注意:形参为数组名时,在被调函数的数组定义中,常采用不定尺寸数组形式,如一维数组的维数、二维数组的第一维可缺省,原因是数组元素的个数在主调函数中是可变化的,而被调函数中的形参不随调用函数实参的变化而改变。

(2)形参为指针变量

当函数调用时,实参数组名的地址值传递给形参指针变量,则形参指针变量就有了指向,它与实参数组名指向同一数组的首地址(如图7.4所示),通过指针变量的移动与取内容运算符,可改变指针变量所指向的间接值,这间接值就是实参数组中的对应元素值。

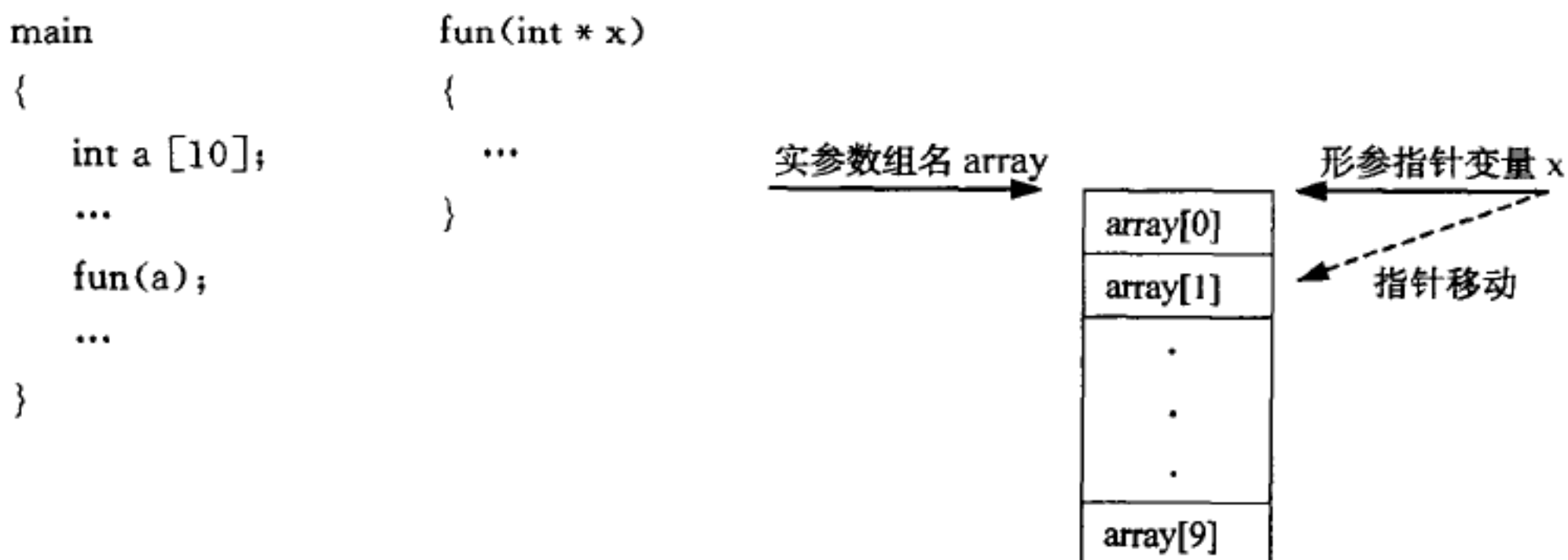


图7.4 实参为数组名,形参为指针变量

[例7.5] 求一个一维数组元素的平均值。

```

#define M 10
#include <stdio.h>
main()
{
    int a[M],i;
    float Avg(int *);
    for(i=0;i<M;i++)
        a[i]=i;
    printf(" Average=%4.2f\n",Avg(a));
}
float Avg(int * x)
{
    int i;
    float avg=0.0;
    for(i=0;i<M;i++,x++)
        avg+=*x;
    avg/=M;
}

```

```
    return avg;
}
```

运行结果:

Average=4.50

上例程序中,我们看到被调函数在主调函数的说明形式 `float Avg(int *)`,它表明形参是用来接受地址值的,因此,凡形参定义为数组名或指针变量的,都可用此形式来说明函数,当然也可用省略的写法 `float Avg()`。我们提倡前一种写法。另外,在主调函数中用 `a[i]` 表示数组元素,在被调函数中指针用 `*x` 与指针移动(`x++`)来表示数组元素。

[例7.6] 求一个二维数组中最大元素值。

```
#include <stdio.h>
float max_value(float (*x)[4])
{
    int i,j;
    float max;
    max=x[0][0];
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            if (max<*(* (x+i)+j))
                max=*(* (x+i)+j);
    return max;
}
main()
{
    static float a[][4]={1,2,3,4,5,6,7,8,9,10,11,12};
    float amax,max_value();
    amax=max_value(a);
    printf("MAX=%4.1f\n",amax);
}
```

运行结果:

MAX=12.0

从上例我们看到,利用二维数组名作为函数的实参时,其调用形式与一维数组名一样,而形参指针变量可用两种形式:一级指针与数组指针变量。上例中就是采用的数组指针的形式,在处理过程中,该指针并没有移动,而是通过行、列变化来取得各元素值的;如果将被调函数写为:

```
float max_value(float *x)
{
    int i,j;
    float max;
    max=*x;
    for(i=0;i<12;i++,x++)
```

```

        if (max < *x)
            max = *x;
    return max;
}

```

则采用的方式是将二维数组降为一维数组处理,虽然二维数组的逻辑结构是二维的,但其二维数组的存储空间仍是一维的一片连续空间,利用指针移动来取得元素值的,因此,将二维数组作为一维数组处理是顺理成章的事。

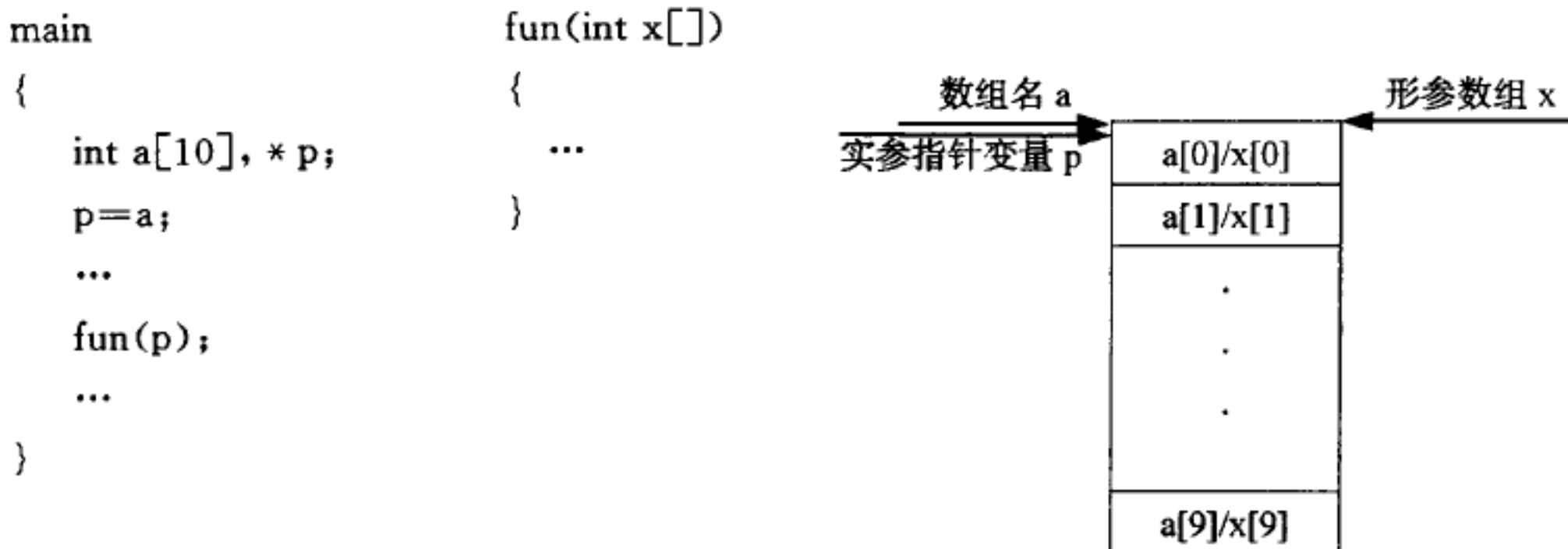
在上两例中,我们采用了函数返回值的形式,在被调函数中,利用“return 变量名/表达式;”语句,就能在主调函数的调用处得到一个由 return 返回的变量名或表达式的值,该值的类型由被调函数的类型决定,这是从被调函数将数据传递给主调函数的另一种方式,它决不是用形参值反传给实参的做法。

2. 以指针变量作为函数的实参

当用指针变量作为函数的实参时,在函数调用之前,该指针变量必须要有指向(即实参在函数调用前要有值),指针变量值可由数组名、字符串、变量地址动态分配得到,而形参只能是数组名与指针变量。

(1) 实参指针变量值指向数组,形参为数组名或指针变量

此种情况与上述以数组名作为函数的实参情况十分相似,只不过现在实参是指针变量,它的值仍然是数组的首地址,即两个数组共享同一存储单元,同时还有一个指针变量指向该数组的首地址,每个元素既可用实参指针变量表示,也可用形参数组名表示(如图7.5所示)。



[例7.7] 求一字符串的长度。

```

#include <stdio.h>
main()
{
    static char a[]="I am student!";
    char *p;
    p=a;
    printf("字符串长度为:%d\n",strl(p));
}
strl(char x[])
{
    int n=0,i;

```

图7.5 实参为指针变量,形参为数组名

```

    for(i=0;x[i]!='\0';i++)
        n+=1;
    return n;
}

```

运行结果:

字符串长度为:13

(2)当实参、形参都为指针变量时,通过函数调用,使两指针指向同一数组的首地址,形参指针变量通过指针移动就可访问数组中的元素(见图7.6)。

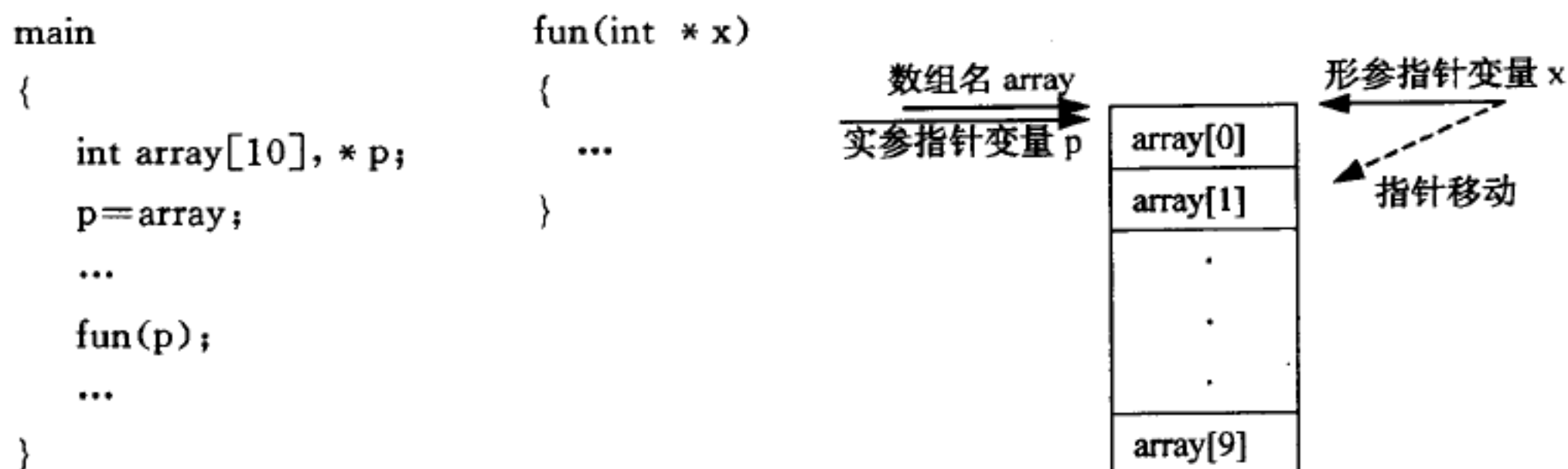


图7.6 实参、形参都为指针变量

[例7.8] 如果从键盘输入字符串 qwerty 和字符串 abcd,求下面程序的运行结果。

```

#include "string.h"
strle(char *s1,char *s2)
{
    int num=0;
    while( *(s1+num)!='\0') num++;    /* num 得到 s1的最后元素位置 */
    while( *s2)
    {
        *(s1+num)= *s2;num++;/* 将 s2的元素一个个地加到 s1的尾上 */
        s2++;
    }
    return num;
}
main()
{
    char str1[15],str2[10], *p1, *p2;
    p1=str1;
    p2=str2;
    gets(p1);
    gets(p2);
    printf("%d\n",strle(p1,p2));
}

```

运行结果:

10

(3) 实参指针变量由系统动态分配指向, 形参为数组名或指针变量

如果在主调函数中定义了一个指针变量, 一旦它指向一个由系统动态分配的地址空间时, 则在被调函数中可用其存储空间存放数据。

[例7.9] 在系统动态分配的存储空间中存放具有10个字符的字符串。

```
#define N 10
#include "stdlib.h"
fun(char a[], int n)
{
    int i;
    for(i=0; i<n-1; i++)
        scanf("%c", a+i);
    a[i]='\0';
}
main()
{
    char *p;
    p=(char *)calloc(N, sizeof(char));
    fun(p, N);
    printf("%s\n", p);
    free(p);
}
```

运行结果:

```
welcome!    /* 用户键盘输入 */
welcome!    /* 输出结果 */
```

上例是处理字符串问题, 其一维字符数组本可用 `scanf("%s", a)` 进行整体输入, 但由于实参指针变量 `p` 的指向是由系统动态分配的, 其存储空间大小已限定, 因此, 形参数组元素个数不能越界, 否则会造成不良影响。

3. 实参为字符串, 形参为数组名或指针变量

由于字符串本身就表示地址, 与数组十分相似, 只是数组以数组名为标志, 因而, 形参须为数组名或指针变量。

[例7.10] 将一字符串的内容颠倒输出。

```
#include <string.h>
main()
{
    fun("abcde");
}
fun(char x[])
{
```



```
    int i,j;
    char k;
    for(i=0,j=strlen(x)-1;i<j;i++,j--)
    {
        k=x[i]; x[i]=x[j]; x[j]=k;
    }
    printf("%s\n",x);
}
```

运行结果:

edcba

如果上例的形参为指针变量,则 fun 函数的定义应为:

```
fun(char *x)
{
    int i,j; char k;
    for(i=0,j=strlen(x)-1;i<j;i++,j--)
    {
        k=* (x+i); * (x+i)=* (x+j); * (x+j)=k;
    }
    printf("%s\n",x);
}
```

其运行结果同上。

4. 实参为变量地址值,形参可为指针变量或一维数组

[例7.11] 求下列程序中函数的递归调用的运行结果。

```
fun(int n,int *x)          /* 也可为 fun(int n, int x[]) */
{
    int f1,f2;
    if(n==1||n==2)
        *x=1;
    else
    {
        fun(n-1,&f1);
        fun(n-2,&f2);
        *x=f1+f2;
    }
}
main()
{
    int a;
    fun(6,&a);
    printf("%d\n",a);
}
```

```
}
```

运行结果:

```
8
```

7.2.3 利用函数返回值和外部变量进行函数数据传递

1. 利用函数返回值进行函数数据传递

要想改变主调函数中某些变量的值,除用7.2.2中所讲述的按址传递方式外,还可用函数返回值的方式,但函数返回值只能是一个,它只能改变主调函数中的一个变量值,并不能改变多个变量值,有关函数返回值的例子在函数章节中已有许多描述,在此,仅举特例说明。

[例7.12]

```
main()
{
    int n1,n2,max;
    scanf("%d%d",&n1,&n2);
    printf("n1的输入值是:%d",n1);
    n1=fun(n1,n2);
    printf("n1重新赋值(函数的返回值)是:%d\n",n1);
}
fun(int x,int y)      /* 此处有函数返回值,缺省则默认为整型 */
{
    if(x>y) return x;
    else return y;
}
```

运行结果:

```
5 12
```

```
n1的输入值是:5
```

```
n1重新赋值(函数的返回值)是:12
```

在上例中,fun 函数是按值传递,n1的值作为实参值传递,不可能变化,但利用函数返回值却可对 n1重新复制赋值,使 n1得到变化,这同一个变量不变和变的两种情况是两个不同的概念,读者一定要加以区分,另外,在被调函数中,有两个 return 语句,但主调函数中只能接受其中的一个。

2. 用外部变量进行函数数据传递

在函数外部定义的变量称外部变量或全局变量,它可被多个函数引用,因而,利用外部变量可进行函数数据传递。外部变量与函数的通信既遵守实参向形参单向传递的原则,同时也有自己的特点。

[例7.13] 外部变量作为函数的参数。

```
int a=3,b=5;
fun(int x,int y)
{
```

```

        x *= y; y += 5;
    }
    main()
    {
        fun(a,b);
        printf("%d,%d\n",a,b);
    }

```

运行结果:

3,5

上例中,外部变量 a、b 的值 3、5 分别传递给函数 fun 中的形参 x、y,而 fun 函数体中对形参值的改变,并不能改变外部变量的值,它依然遵循单向传递原则。

[例 7.14] 利用被调函数给外部变量赋值。

```

int a,b;
fun(int x,int y){ a=y;b=x*a;}
main()
{
    int c=3,d=5;
    fun(c,d);
    printf("%d,%d\n",a,b);
}

```

运行结果:

5,15

上例中,实参 c、d 的值 3、5 分别传递给函数 fun 中的形参 x、y,在 fun 函数体中利用形参的值对外部变量赋值,使外部变量有了如上的结果。利用这一办法,我们可以得到多个值的反传,这是由于外部变量的作用域是从定义处开始,直到整个文件结束。

7.2.4 结构作为函数参数传递

1. 结构与数组

结构与数组的含义十分相似,前者是描述不同类型数据的集合,用一整体表示,而后者则是相同类型数据的集合,也用一整体表示,它们的数据类型同属构造型(见表 7.1)。

表 7.1 结构、数组与结构数组的比较

名称	类型表示	组成	存储空间大小	存储空间分配时间
结构	struct...	成员	\sum 成员字节	定义结构变量时
数组	...[]	元素	sizeof(数据类型) * 元素个数	定义数组时
结构数组	struct...[]	结构	\sum 成员字节 * 元素个数	定义结构数组时

两者虽有相似之处,但作为函数参数的传递却有很大差别。

2. 利用结构作为函数参数传递

(1) 用结构变量 作为函数的参数

[例7.15]

```
#include "stdio. h"
struct st
{
    int a;
    char b;
};
fun(struct st bc)
{
    bc. a += 5;
    bc. b = 'A';
    printf("被调函数中成员的值是:%d, %c\n", bc. a, bc. b);
}
main()
{
    struct st bl;
    bl. a = 3;
    bl. b = 'c';
    fun(bl);
    printf("主调函数中成员的值是:%d, %c\n", bl. a, bl. b);
}
```

运行结果:

被调函数中成员的值是:8,A

主调函数中成员的值是:3,c

在上例中,利用结构变量作为函数的参数传递,虽然合法,但传递过程中所有成员值都必须传递,既费时又开销大,特别是成员值多时,程序运行效率明显下降。因此,采用结构指针作为函数参数比较适用。同时,被调函数中成员值不会改变主调函数中成员值,与一般按值传递相同;若用结构变量地址作为实参值,则与按址传递相同。

[例7.16]

```
#include "stdio. h"
struct st
{
    int a;
    char b;
};
fun(struct st * bp)
{
```

```

    bp->a+=5;
    bp->b='A';
    printf("%d, %c\n", bp->a, bp->b);
}
main()
{
    struct st bl;
    bl.a=3;
    bl.b='c';
    fun(&bl);
    printf("%d, %c\n", bl.a, bl.b);
}

```

运行结果:

8, A

8, A

[例7.17]

```

#define N 3
struct st
{
    int num;
    char name[7];
    float score;
};
main()
{
    int i;
    float avg, inpu _ avg();
    struct st student[N];
    avg=inpu _ avg(student); /* 结构数组名作为函数的实参 */
    printf("学号 姓名 成绩\n");
    for(i=0;i<N;i++)
        printf("%4d%7s%6.1f\n", student[i].num, student[i].name,
            student[i].score);
    printf("平均成绩:%4.1f\n", avg);
}
float inpu _ avg(struct st *p)
{
    int i;
    float temp, avg=0.0;
    for(i=0;i<N;i++)

```



```

    {
        scanf("%d%s%f", &(p+i)->num, (p+i)->name, &temp);
        (p+i)->score = temp;
        avg += (p+i)->score;
    }
    avg = avg/N;
    return avg;
}

```

运行结果:

```

1 王 军 78.0
2 吴晓兰 67.0
3 张 东 92.5 /* 以上是键盘输入 */
学号 姓名 成绩
1 王 军 78.0
2 吴晓兰 67.0
3 张 东 92.5
平均成绩:79.2

```

上例利用结构指针数组名作为实参值传递给形参结构指针变量,通过结构指针变量的下标值变化,对结构指针数组各元素赋值。很明显,结构成员并不是传递的对象,传递的仅是地址值,因而,如同数组一样,在主调函数中我们很容易得到多个结构数组的元素值。值得注意的是,在 Turbo C 系统中,scanf 函数对构造型的 float 型数据,如实型的二维数组元素及结构成员,在连接时会产生 float 格式错误,因此,采用例中的 temp 中间变量过渡是一权宜之计,在其他系统中不会出现这样的问题。

7.3 函数指针与指针函数

7.3.1 函数指针

1. 定义

指针变量可以指向变量地址、数组、字符串、动态分配地址,同时也可指向函数,每一个函数在编译时,系统会分配给该函数一个入口地址,函数名表示这个入口地址,那么,指向函数的指针变量称之函数指针变量,例如:

```

...
float fun(int, char), (*p)();
p = fun;
...

```

表7.2总结了指向变量地址、数组名、函数名的指针变量的一些性质。

表7.2 指向变量地址、数组名、函数名的指针变量

名称	形式	指针变量		指向	指针的类型	指针的 增1运算
		定义	名称			
变量	i	*p	一般指针变量	p=&i	与i相同	无
数组	a[][]	(*p)[]	数组指针变量	p=a	与数组元素相同	有
函数	fun()	(*p)()	函数指针变量	p=fun	与函数返回值相同	无

2. 用函数指针变量调用函数

前面已描述了一般的函数调用形式:函数名(<实参表>),现也可用函数指针来调用函数,其形式为:(*函数指针变量名)(<实参表>)

例如:

<pre>... int i=5; char ch='a' float fun(int,char),(*p)(); p=fun; (*p)(i,ch); ...</pre>	<pre>... int i=5; char ch='a'; float fun(int,char); fun(i,ch); ...</pre>
--	--

上例的左右两边的函数调用形式是等价的,但有多函数调用时,左边用函数指针变量调用的方式要比右边用函数名调用方便与简洁,因为函数指针变量是存放函数的入口地址,它可以不固定指向某个函数,需要调用某函数时,就将某函数名赋给该函数指针变量,当用函数指针变量调用该函数时,即可转向执行该函数。

[例7.18] 利用直接函数调用,求函数返回值。

```
#include <stdio.h>
main()
{
    int a=3,b=5;
    float f1(),f2();
    printf("%4.1f\n",f1(a,b));
    printf("%4.1f\n",f2(a,b));
}
float f1(int x,int y)
{
    return x+y;
}
float f2(int x,int y)
{
    return (1.0*y)/x;
```

}

运行结果:

8.0

1.7

[例7.19] 利用函数指针变量调用,求函数返回值。

#include <stdio.h>

```

      a      b      f1/f2      /* 即 fun=f1或 fun=f2 */
      ↓      ↓      ↓
sub(int x,int y,float (*fun)())

```

{

float result;

result = (*fun)(x,y); /* 等价于调用 f1(x,y) 或 f2(x,y) */

printf("%4.1f\n", result);

}

```

      x      y
      ↓      ↓
float f1(int x1,int y1)

```

{

return x1+y1;

}

```

      x      y
      ↓      ↓
float f2(int x2,int y2)

```

{

return (y2 * 1.0)/x2;

}

main()

{

int a=3,b=5;

float f1(),f2();

sub(a,b,f1);

sub(a,b,f2);

}

运行结果同上。

在主调函数中调用多个类似的函数时,我们可用上述两种方法处理,[例7.18]采用函数直接调用形式,而[例7.19]则采用函数指针变量的调用形式,它利用函数名作为函数的实参,函数指针变量作为形参(接受函数的入口地址),若有更多的函数要调用,我们只要在适当的位置加入各函数的定义,并在主调函数中加入对应的 sub 函数即可,而 sub 函数并不作任何变更,它是一种结构化程序设计方法。

7.3.2 指针函数

1. 定义

函数返回值可以是 int、char、float 等,也可以为地址值。函数返回值是地址值的函数称指

针函数,因此,其函数返回值必须用同类型的指针变量来接受,也就是说,指针函数一定有函数返回值,而且,在主调函数中,函数返回值必须赋给同类型的指针变量。例如:

<数据类型> * 函数名(),指针变量; 指针变量=函数名(实参表);	float * fun(), * p; p=fun(a);
--	----------------------------------

[例7.20] 指针函数的含义。

```
main()
{
    float a[]={1,2,3,4,5}, * p, * fun();
    p=fun(a);                /* 指针函数的调用形式 */
    printf("%3.1f\n", * p);
}

float * fun(float * x)
{
    return ++x;
}
```

运行结果:

2.0

上例是一个简单例子,它将数组名传递给一指针变量,然后,指针变量移动指向下一个数组元素,再将此地址返回给 p 指针,因此, p 指针就指向下一个元素。

2. 示例

[例7.21] 两个字符串的连接。

```
#include "string.h"
#define N1 20
#define N2 10
main()
{
    char * s,s1[N1],s2[N2];      /* s 指向 s1+s2 */
    char * strsl2(char *,char *);
    gets(s1);
    gets(s2);
    s=strsl2(s1,s2);             /* 两字符串连接后的首地址 */
    printf("s12=%s\n",s);
}

char * strsl2(char * p1,char * p2)
{
    char * temp;
    temp=p1;
    while(* p1)                  /* 指针 p1移动到所指向的字符串末尾 */
        p1++;
    return temp;
```

```

while( * p2)           /* 将 p2指向的字符逐个追加到 p1的末尾 */
{
    * p1 = * p2;
    p1++;p2++;
}
p1='\0';
return temp;           /* 返回指针 p1的首地址 */
}

```

运行结果:

```

I am a good
student           /* 以上是键盘输入 */
I am a good student /* 输出结果 */

```

7.4 数组指针、指针数组与带参的 main 函数

7.4.1 数组指针

数组指针问题在指针一章中已讨论过,它是一个行指针,用来指向二维数组,其定义为:

<数据类型> (* 指针变量名)[二维数组的列数]

它可指向二维数组的任一行地址,例如:

```

int (* p)[4],a[3][4];
p=a;           /* 指向二维数组的首地址,即0行地址,
                若 p=a[1],则指向1行地址 */

```

因此,p++/p--均表示行地址的移动,即每次移动列数,有关详细内容不在此讨论。

7.4.2 指针数组

1. 定义

凡数组元素是指针类型数据的数组称之为指针数组,其定义为:

<数据类型> * 数组名[元素个数]

例如:

<pre> int * array[3]; int a[3]; array[0]=&a[0]; array[1]=&a[1]; array[2]=&a[2]; </pre>	<pre> char * array[3]; array[0]="for"; array[1]="while"; array[2]="if"; </pre>
--	--

左边的指针数组是整型的,它的每个元素都是地址值,即一维整型数组 a 各元素的地址。右边

的指针数组是字符型的,从外表看指针数组好似一维数组,但它却包含着二维数组的内容,其每个元素都是指针、即不同长度字符串的首地址。

2. 一维字符指针数组与二维字符数组的区别

首先,我们写出如下二维字符数组与一维字符指针数组的初始化形式(省略 static 说明):

```
char a[][6]={"for","while","if"}; | char *array[]={"for","while","if"};
```

联系定义中的形式,从中我们可以看出它们之间的区别:

(1)二维字符数组的行地址是指针常量,不能对其重新赋值,而一维字符指针数组的数组元素是指针变量,它可取得各种地址值:数组名、字符串等。

[例7.22]

```
main()
{
    char array[3][6];
    int i;
    strcpy(array[0],"for");
    strcpy(array[1],"while");
    strcpy(array[2],"if");
    for(i=0;i<3;i++)
        printf("%s\n",array[i]);
}
```

运行结果:

```
for
while
if
```

```
main()
{
    char *array[3];
    int i;
    array[0]="for";
    array[1]="while";
    array[2]="if";
    for(i=0;i<3;i++)
        printf("%s\n",array[i]);
}
```

运行结果与左相同。

(2)二维字符数组中的一维数组元素是等长的,一维字符指针数组的数组元素是不等长的,当处理不等长字符串时,二维字符数组所占用的存储空间大(以最长字符串的长度为各元素存储空间),一维字符指针数组是按各字符串的实际长度来分配存储空间的,因此,二维字符数组用来处理等长字符串问题,一维字符指针数组则处理不等长字符串问题。

7.4.3 带参的 main 函数

我们通常使用的主函数的形式为 main(),它是一种不带参与无返回值的形式,我们也可写为 void main(void),现我们描述它带参的情况。

1. 命令行参数

在操作系统下为执行某个程序或命令而键入的一行字符称命令行,通常命令行含有可执行文件名及若干个参数,并以回车结束。如:

```
C:\>copy myfile.c file.c
```

其中 C:\>是操作系统的提示符,而 copy myfile.c file.c 是命令行的三个参数,参数间用空格隔开,三个参数一般是三个不等长的字符串,处理这些参数用指针数组比用二维数组会更简洁、方便,这也是一维字符指针数组的一种重要应用。

2. main 函数的参数

当 main 函数带参时,它的形式为:main(int argc,char * argv[]),其中 argc 与 argv 是 main 函数的两个形参。main 函数是由系统调用,形参的值是由命令行参数给出,形参 argc 是统计命令行参数的个数,所以它是整型数据,形参 argv 是指针数组,它的每个元素指向命令行对应以字符串表示的参数,其元素个数由 argc 确定。

[例7.23] 下列程序的可执行文件名为 test.exe,在该程序中,main 函数的参数在内存中的存储情况见图7.7所示。

```
main(int argc,char * argv[])
{
    int i;
    printf("argc=%d\n",argc);
    for(i=1;i<argc;i++)
        printf(" %s\n",argv[i]);
}
```

argc	3
argv[0]	test\0
argv[1]	IBM-PC\0
argv[2]	COMPUTER\0

图7.7 main 函数的参数

在操作系统提示符下,输入如下命令行:

C:\>test IBM-PC COMPUTER ✓

则程序运行结果:

```
argc=3
IBM-PC
COMPUTER
```

[例7.24] 下列程序的可执行文件名为 sm.exe,该程序根据命令行参数实现求一个正整数的累加和。

```
#include "stdlib.h"
main(int argc,char * argv[])
{
    int n,i,s=0;
    n=atoi(argv[2]);
    if(argc<3) exit(0);
    for(i=1;i<=n&&*argv[1]=='+';i++)
        s+=i;
    printf("sum=%d\n",s);
}
```

在操作系统提示符下,输入如下命令行:

C:\>sm + 10 ✓

则程序运行结果:

```
sum=55
```

7.5 单向链表

7.5.1 单向链表的概念

在现实生活中,我们经常接触到一些链状的东西,如项链、自行车的传动链条等,它们都是由一些链环与关联部件组成,在计算机中则称之为链表,其中各链环称为结点,关联部件则是与结点相连的指针变量。由于连结的方式不同,链表的名称也不同,本书仅讲述一个结点与后一个结点连结的单向链表。

1. 链表是结构、指针相结合的一种应用,它是由头、中间、尾多个链环组成的单方向可伸、缩的链表,链表上的链环我们称之为结点。

2. 每个结点的数据可用一个结构体表示,该结构体由两部分成员组成:数据成员与结构指针变量成员。

3. 数据成员存放用户所需数据,而结构指针变量成员则用来连接(指向)下一个结点,由于每一个结构指针变量成员都指向相同的结构体,所以该指针变量称为结构指针变量。

4. 链表的长度是动态的,当需要建立一个结点,就向系统申请动态分配一个存储空间,如此不断地有新结点产生,直到结构指针变量指向为空(NULL)。申请动态分配一个存储空间的表示形式为:

```
(struct note *)malloc(sizeof(struct note))
```

5. 一个简单结点的结构体表示为:

```
struct note
{
    int data;                /* 数据成员可以是多个不同类型的数据 */
    struct note * next;      /* 指针变量成员只能是一个 */
};
```

6. 一个简单的单向链表的图示(见图7.8):

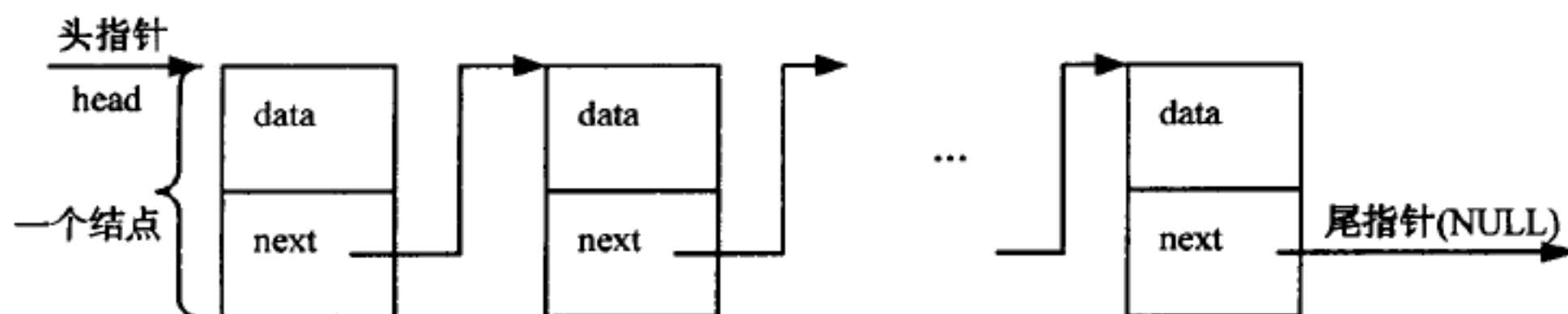


图7.8 一个简单的单向链表

7.5.2 链表的建立

在链表建立过程中,首先要建立第一个结点,然后不断地在其尾部增加新结点,直到不需

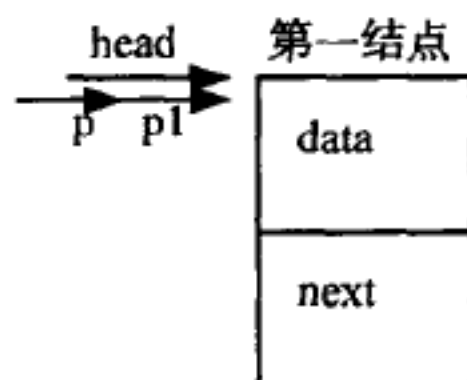
再有新结点,即尾指针指向 NULL 为止。

设有结构指针变量 struct note p, pl, head;, head 指针变量用来标志链表头,在链表建立过程中, p 总是不断先接受系统动态分配给新结点的地址,用 $p1 \rightarrow next$ 存储新结点的地址,那么,新结点就和上一个结点连接起来,如此周而复始,既与递归相似又像循环。

链表建立步骤:

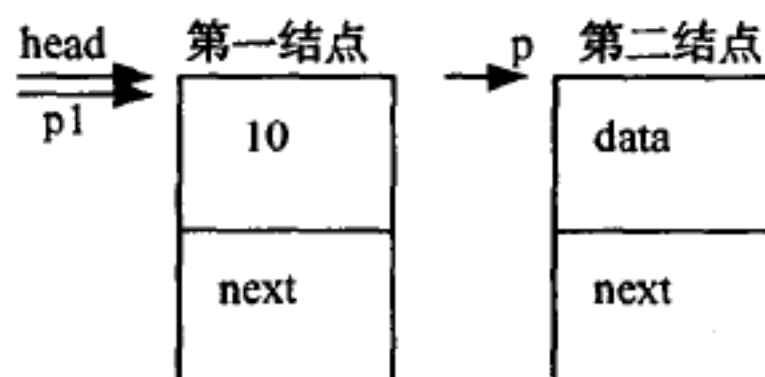
第一步:建立第一个结点。

```
struct node
{
    int data;
    struct note * next;
};
struct note p, p1, * head;
head=p1=p=(struct node *)malloc(sizeof(struct node));
```



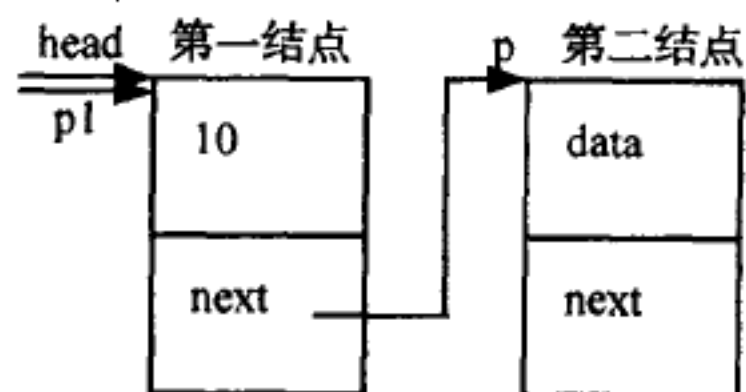
第二步:给第一个结点成员 data 赋值并产生第二结点。

```
scanf("%d",&p->data); /* 输入10 */
p=(struct node *)malloc(sizeof(struct node));
```



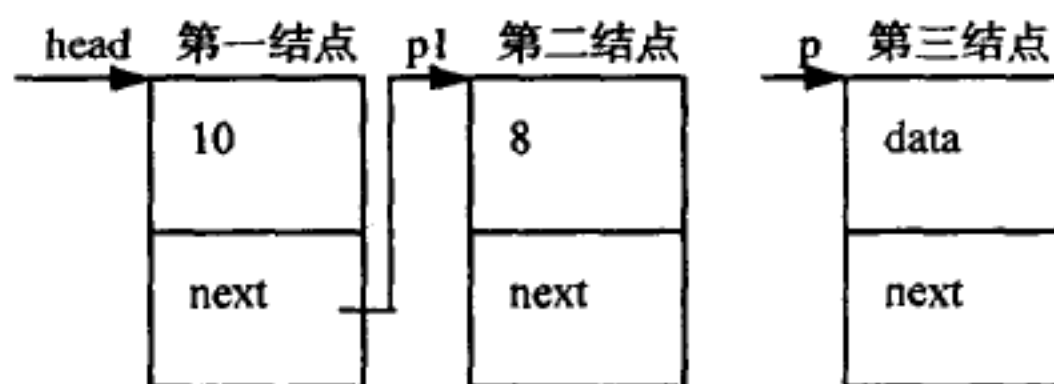
第三步:将第一、二结点连接起来。

```
p1->next=p;
```



第四步:产生第三个结点。

```
p1=p;
scanf("%d",&p->data); /* 输入8 */
p=(struct node *)malloc(sizeof(struct node));
```



以后步骤都是重复第三、四步,直到给出一个结束条件,不再建新的结点时,要有 $p \rightarrow \text{next} = \text{NULL}$; ,它表示尾结点。

[例7.25] 建立链表。

```
#include "stdio.h"
#include "alloc.h"
#define LEN sizeof(struct node)
struct node
{
    int data;
    struct node * next;
};
main()
{
    struct node * p, * pl, * head;
    head = p = (struct node *) malloc(LEN);
    scanf("%d", &p->data); /* 头结点的数据成员 */
    while(p->data != 0) /* 给出0结束条件,不再建新的结点,退出循环 */
    {
        pl = p;
        p = (struct node *) malloc(LEN);
        scanf("%d", &p->data); /* 中间结点的数据成员值 */
        pl->next = p;          /* 中间结点的指针成员值 */
    }
    p->next = NULL;           /* 链表建立已结束,尾结点的指针成员值 */
    ...
}
```

为了证实已建链表是所需要的,应在上程序的省略处加入下列程序段:

```
...
p = head;          /* 链表显示 */
printf("链表数据成员是:");
while(p->next != NULL)
{
    printf("%d ", p->data);
    p = p->next;
}
printf("%d\n", p->data);
...
```

运行结果:

```
10 8 6 4 2 0          /* 建链表时输入的数据 */
链表数据成员是: 10 8 6 4 2 0    /* 显示所建的链表 */
```


7.5.3 链表结点的删除

1. 概述

链表结点的删除意味着必须将某个要删除结点前后的连接打断,去掉该结点,使前后指针变量重新连接,完成链表结点的删除任务。

如图7.9所示,在上述所建链表中,假定删除第二个结点,就必须将第一个结点的结构指针成员指向第三个结点,则第二个结点被删除(如图中虚线所示)。

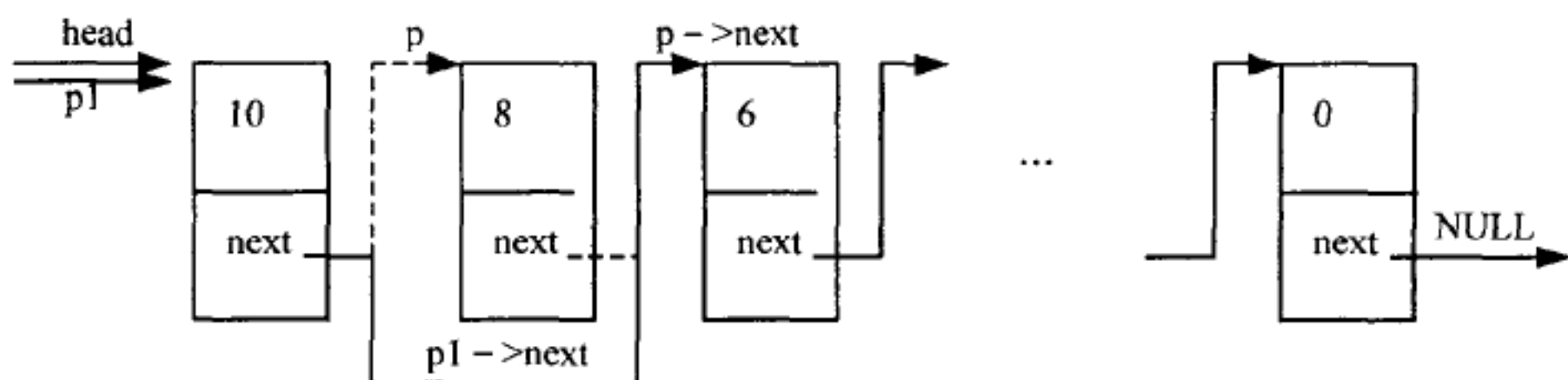


图7.9 链表结点的删除

2. 示例

[例7.26] 编写实现图7.9所示链表结点删除的程序。

(在上述的链表建立与显示程序的最后加入下列程序段)

```
...
p=head;
while(p->next!=NULL)
{
    if (p->data==8)
        p1->next=p->next;
    p1=p;
    free(p);          /* 释放所删除结点的空间 */
    p=p1->next;
}
p=head;
printf("\n 删除数据成员为8的结点后的链表数据是:");
while(p->next!=NULL)
{
    printf(" %d ",p->data);
    p=p->next;
}
printf(" %d\n",p->data);
...
```

运行结果:

```
10 8 6 4 2 0          /* 键盘输入链表各结点数据 */
链表数据成员是: 10 8 6 4 2 0
删除数据成员为8的结点后的链表数据是: 10 6 4 2 0
```

7.5.4 链表结点的插入

1. 概述

链表结点的插入意味着要在某结点前或后插入一个或多个结点, 所插入的结点必须由系统重新动态分配地址 $p2 = (\text{struct node } *) \text{malloc}(\text{LEN})$; (指针变量 $p2$ 指向该地址), 然后, 将原链表插入处前结点 $p \rightarrow \text{next}$ 取出存放在指针变量 $p3$ 中, 即 $p3 = p \rightarrow \text{next}$, 再将新结点的地址赋给而原插入处前结点的 $p \rightarrow \text{next}$, 即 $p \rightarrow \text{next} = p2$, 而原插入处后结点的地址值 ($p3$) 赋给新结点的 $p2 \rightarrow \text{next}$, 即 $p2 \rightarrow \text{next} = p3$ (见图 7.10)。

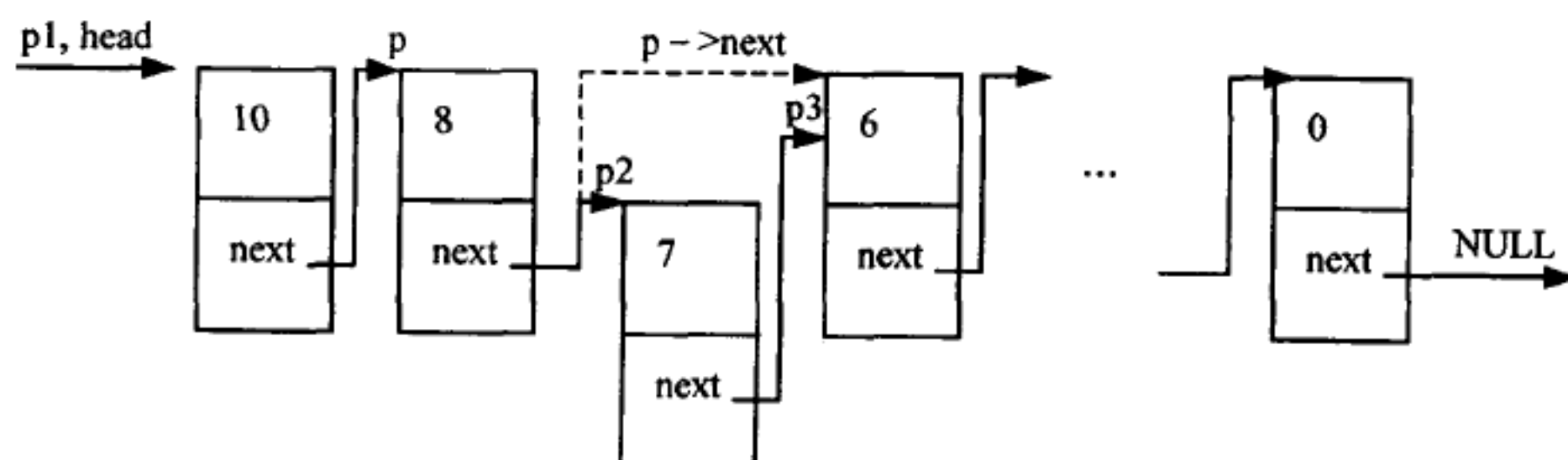


图 7.10 链表结点的插入

2. 注意

(1) 本节仅描述在某结点后插入, 若想在某结点之前插入, 其原理是一样, 读者可自行设计。

(2) 在插入操作中, 多增加了两个结构指针变量 $p2$ 、 $p3$ 。

3. 示例

[例 7.27] 在图 7.10 链表中插入新结点的程序段。

```
...
p = head;
while (p->next != NULL)
{
    p1 = p;
    p = p1->next;
    if (p->data == 8)
    {
        p3 = p->next;
        p2 = (struct node *) malloc(LEN);
        scanf("%d", &p2->data);
        p->next = p2;
```

```

    p2->next=p3;
}
}
...

```

运行结果:

```

10 8 6 4 2 0      /* 原链表数据成员 */
7                /* 输入新结点的数据成员 */
10 8 7 6 4 2 0    /* 在数据成员8结点后插入数据成员为7的结点 */

```

[例7.28] 单向链表中任意位置插入任意多个新结点。

```

#include "stdio.h"
#include "alloc.h"
#define LEN sizeof(struct node)
struct node
{
    int data;
    struct node * next;
};
main()
{
    struct node * p, * p1, * p2, * p3, * head;
    int i;
    head=p=(struct node *)malloc(LEN); /* 链表建立 */
    scanf("%d",&p->data);
    while(p->data!=0)
    {
        p1=p;
        p=(struct node *)malloc(LEN);
        scanf("%d",&p->data);
        p1->next=p;
    }
    p->next=NULL;
    p=head;          /* 插入一个或多个新结点 */
    while(p->next!=NULL)
    {
        printf("\n 在带有%d 数据成员的结点后插入吗:",p->data);
        i=getche();
        if(i=='y')
        {
            p3=p->next;
            p2=(struct node *)malloc(LEN);

```

```

        scanf("\n 输入新结点的数据%d: ", &p2->data);
        p->next=p2;
        p2->next=p3;
    }
    p1=p;
    p=p1->next;
}
p=head;          /* 新结点插入后的链表数据成员显示 */
printf("\n");
while(p->next!=NULL)
{
    printf(" %d ", p->data);
    p=p->next;
}
printf(" %d\n", p->data); /* 新结点插入后的链表数据成员显示 */
}

```

运行结果:

10 8 6 4 2 0

在带有10数据成员的结点后插入吗:n

在带有8数据成员的结点后插入吗:y

输入新结点的数据:7

在带有7数据成员的结点后插入吗:n

在带有6数据成员的结点后插入吗:y

输入新结点的数据:5

在带有5数据成员的结点后插入吗:n

在带有4数据成员的结点后插入吗:n

在带有2数据成员的结点后插入吗:n

10 8 7 6 5 4 2 0 /* 带下划线的数据为新插入结点的数据成员 */

7.6 小 结

本章是在讲授数组、指针、函数基本概念与内容之后的综合应用的章节,它从应用的角度将上述内容糅合在一起。我们讲述了将数组、指针变量、字符串、结构、函数作为函数的参数进行数据的传递,并用实例体现其应用。函数参数小结见表7.3。

表7.3 函数参数小结

参数名	形 式		作 用
	实 参	形 参	
变量 int x;	fun(x);	fun(int y)	x 的值传给 y, x、y 值相互独立。
	fun(&x)	fun(int *p1) *p1=y;	x 的地址值传给 p, y 的值影响 x 的值。
数组名 int array[N];	fun(array);	fun(int array1[]) fun(int *p1)	传递数组首地址, 可改变数组元素的值。
指针变量 int *p, array[N]; p=array;	fun(p);	fun(int array1[]) fun(int *p1)	传递指针变量所指向的数组首地址。
字符串 "abcd"	fun("abcd");	fun(char array1[]) fun(char *p1)	传递字符串的首地址。
结构变量 struct student x;	fun(x);	fun(struct student y)	x 的结构成员数据传给 y 的对应成员, 与一般变量的传递一样。
	fun(&x)	fun(struct student *p1)	结构变量的地址传递给结构指针变量, 利用间接赋值可改变结构成员的值。
函数名 int f1();	fun(f1)	fun(int (*p1)()) ... (*p1)(x,y);	函数指针变量指向函数 f1; 等价于 f1(x,y)。

习 题

一、选择题

7.1 若有定义 `int (*p)[3];` 和 `int (*f)();`, 则 p 与 f 分别是【1】。(数组指针, 函数指针)

- 【1】 A) 指针数组名、函数指针名 B) 数组指针名、指针函数名
C) 数组指针名、函数指针名 D) 指针数组名、指针函数名

7.2 若有定义 `char *language[]={ "FORTRAN", "BASIC", "PASCAL", "JAVA", "C" }`; , 则 `language[2]` 的值是【2】。(指针数组)

- 【2】 A) 一个字符 B) 一个地址
C) 一个字符串 D) 不定值

7.3 若有函数 `max(a,b)`, 为了让函数指针变量 p 指向函数 max, 正确的赋值语句是【3】。(函数指针)

- 【3】 A) `p=max;` B) `*p=max;`
C) `p=max(a,b);` D) `*p=max(a,b);`

7.4 下面程序的运行结果是【4】。(变量地址, 指针变量)

```
#include <stdio.h>
void sub( int *x, int y, int z )
```



```

{
    *x = y - z;
}
main()
{
    int a, b, c;
    sub( &a, 10, 5 );
    sub( &b, a, 7 );
    sub( &c, a, b );
    printf( "%d,%d,%d\n", a, b, c );
}

```

【4】 A) 10, -2, 5

B) 10, 5, 7

C) 10, -2, 7

D) 5, -2, 7

7.5 下面程序的输出结果为【5】。(静态, 无参)

```

#include <stdio.h>
f()
{
    static c = 3;
    c++;
    return(c);
}
main()
{
    int i, k;
    for( i = 0; i < 2; i++ ) k = f();
    printf( "%d\n", k );
}

```

【5】 A) 3

B) 4

C) 5

D) 6

7.6 下面程序的输出结果为【6】。(实参为字符串常量, 形参为指针变量)

```

#include <stdio.h>
int f(char *s)
{
    char *p = s;
    while( *p != '\0' ) p++;
    return (p - s);
}
main()
{
    printf( "%d\n", f("FUJIAN") );
}

```

【6】 A)0

B)6

C)7

D)8

7.7 下面程序的输出结果为【7】。(动态分配)

```
#include <stdio.h>
#include <alloc.h>
fun(int *s, int b[][3])
{
    *s=b[1][1];
}
main()
{
    int a[][3]={1,3,5,7,9,11}, *p;
    p=(int *)malloc(sizeof(int));
    fun(p,a);
    printf("%d\n", *p);
}
```

【7】 A)7

B)1

C)3

D)9

7.8 下面程序的输出结果为【8】。(结构变量地址,结构数组传递)

```
#include <stdio.h>
main()
{
    void func();
    struct date
    {
        int a;
        char s[5];
    }arg;
    arg.a=27;
    strcpy(arg.s, "abcd");
    func(&arg.a, arg.s);
    printf("arg.a=%d, arg.s=%s\n", arg.a, arg.s);
}
void func(int *x, char s1[])
{
    *x-=5;
    strcpy(s1, "ABCD");
}
```

【8】 A)arg.a=22,arg.s=ABCD

B)arg.a=27,arg.s=abcd

C)arg.a=22,arg.s=abcd

D)arg.a=27,arg.s=ABCD

7.9 下面程序的运行结果是【9】。(数组名,指针变量)

```
#include <stdio.h>
void fun(char *s);
```

```

main()
{
    static char str[]="123";
    fun(str);
}
void fun( char *s )
{
    if( *s )
    {
        fun( ++s );
        printf( "%s\n", --s );
    }
}

```

- 【9】 A)3 B)123 C)1 D)3
- 32 12 12 23
- 321 1 123 123

7.10 下面程序的运行结果是【10】。(指针变量,数组)

```

#include <stdio.h>
void fun( char s1[])
{
    int i, j;
    for( i=j=0; *(s1+i)!='\0'; i++ )
        if( *(s1+i) < 'n' )
        {
            *(s1+j) = *(s1+i);
            j++;
        }
    *(s1+j) = '\0';
}
main()
{
    char str[]="morning", *p;
    p=str;
    fun(p);
    puts(p);
}

```

- 【10】A)morig B)morning C)mig D)or

7.11 下面程序的运行结果是【11】。(数组名,数组名)

```

#include <stdio.h>
void f( int b[] )

```

```

{
    int i=0;
    while( b[i]<=10 )
    {
        b[i]+=2;
        i++;
    }
}
main()
{
    int i,a[]={ 1, 5, 10, 9, 13, 7 };
    f( a+1 );
    for( i=0; i<6; i++ )
        printf( " %d ", a[i] );
}

```

【11】A)2 7 12 11 13 9

B)1 7 12 11 13 7

C)1 7 12 11 13 9

D)1 7 12 9 13 7

7.12 下面程序的运行结果是【12】。(指针,指针)

```

#include <stdio. h>
fun(char *s)
{
    char t;
    if( *s)
    {
        t= *s++;
        fun(s);
    }
    if(t!='\0') putchar(t);
}
main()
{
    char *a="1234";
    fun(a);
    printf("\n");
}

```

【12】A)4321

B)1234

C)1324

D)4231

7.13 下面程序的运行结果是【13】。(外部变量)

```

int d=1;
fun(int y)
{

```

```

    int d=5;
    d+=y++;
    printf("%d ",d);
}
main()
{
    int x=3;
    fun(x);
    d+=x++;
    printf("%d\n", d);
}

```

【13】A)9 4 B)9 6 C)8 4 D)8 5

7.14 下面程序的输出结果是【14】。(函数递归调用)

```

#include <stdio.h>
int i;
main()
{
    int i=1,j=2;
    fun( fun( i, &j ), &j );
}
fun( int a, int *b )
{
    static int m=2;
    i+=m+a;
    m=++(*b);
    printf( "%d,%d\n", i, m );
    return(m);
}

```

【14】A)3,3 B)3,3 C)3,3 D)3,3
 6,4 6,3 9,3 9,4

二、填空题

7.15 下面程序输出一个如下所示的5×5的矩阵。(二维数组,数组指针)

```

1   2   3   4   5
10  9   8   7   6
11  12  13  14  15
20  19  18  17  16
21  22  23  24  25
#define N 5
void initial(int 【15】)
{

```



```

    int i, j;
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            if(i%2==0)
                a[i][j]=i * N+j+1;
            else
                a[i][j]=i * N+N-j;
}
main()
{
    int a[N][N];
    int i, j;
    initial(a);
    for(i=0; i<N; i++)
    {
        for(j=0; j<N; j++)
            printf(" %3d ", a[i][j]);
        printf("\n");
    }
}

```

7.16 以下程序的 fun 函数用于统计字符串 s 中元音字母(a、A、e、E、i、I、o、O、u、U)的个数。

```

#include <stdio.h>
main()
{
    char str[255];
    gets(str);
    printf("元音字母的个数为: %d\n", fun(str));
}
fun(char s[16])
{
    char a[]="aAeEiIoOuU", *p;
    int n=0;
    while(*s)
    {
        for(p=a; *p; p++)
            if(*s==*p)
            {
                n++;
                break;
            }
    }
}

```

```
        s++;  
    }  
    return n;  
}
```

- 7.17 以下程序经过编译连接后得到的可执行文件名为 echo.exe,若在 dos 提示符下输入【17】,则在屏幕上将显示 My computer。(命令行参数)

```
main(int argc,char * argv[])  
{  
    int i;  
    for(i=1;i<argc;i++)  
    {  
        printf(" %s%c",argv[i],(i<argc-1)?" ":"\n");  
    }  
}
```

- 7.18 以下程序段用以统计链表中结点的个数。其中 head 指向链表首结点,count 用来统计结点个数。

```
struct node  
{  
    char data;  
    struct node * next;  
};  
main()  
{  
    struct node * p, * head;  
    int count=0;  
    p=head;  
    while(p!=NULL)  
    {  
        count++;  
        p=【18】;  
    }  
}
```

第 8 章

文 件

文件及其操作在程序设计中是非常重要的内容,合理地对其进行利用,可以大大扩展程序的应用范畴和功能。在计算机系统中,文件是一种宝贵的资源和手段,例如我们编写的源程序就需要以文件的形式保存起来,以便能在不同的时间和地点重复地利用。在程序中使用文件操作,可以对文件进行加工处理,或者创建新的文件,使程序的数据得以永久地保存及再利用。

本章建议课堂讲授 3 学时,上机 2~4 学时,自学 4 学时。

8.1 文件、流和文件系统

计算机系统上的文件是一组有序的数据集合,存储在外部介质上(如磁盘等),通过文件名来存取,由操作系统来管理。现代操作系统把所有外部设备认为是文件,以便进行统一的管理。C 语言也是这样,可以认为文件是磁盘文件和其他具有输入输出(I/O)功能的外部设备(如键盘、显示器等)的总称,在这里,文件已是一个逻辑概念,撇开了具体设备的物理形态而只关心其 I/O 功能。

根据数据的组织形式,文件可以分为 ASCII 文件和二进制文件。ASCII 文件又称为文本文件,其相应数据是以字符形式存放,每一个字符用一个 ASCII 字符代码表示,而一个 ASCII 字符代码用一个字节存放,如整数 123 存放在 ASCII 文件中将是'1'、'2'、'3'的字符序列,占 3 个字节的存储空间,'1'的 ASCII 字符代码是 49,即 00110001。

0	0	1	1	0	0	0	1
0	0	1	1	0	0	1	0
0	0	1	1	0	0	1	1

图 8.1 整型数 123 以 ASCII 字符代码存放

二进制文件则是以字节为单位存放数据的二进制代码,它将数据按内存中的形式原样保存在文件之中,如整数 123 存放在二进制文件中,它的二进制数 0000000001111011,其存放形式为:

0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

图 8.2 整型数 123 以二进制数存放

占一个整数的存储空间,即两个字节。ASCII 文件便于进行阅读,如源程序文件,而二进制文件便于计算机直接处理,如执行文件。

与其他高级语言不同,ANSI C 在文件的基础上进一步使用了流的概念,我们可以简单理解流为无结构的连续字节序列,其内容是对文件中数据的一个映射,相应文件的 I/O 操作对于编程者而言都是对流进行的,程序员只需与流打交道,写数据到流中和从流中读取数据。

相似于文件类型,流也有两种:文本流和二进制流。文本流是一行行的字符,换行符'\n'表示这一行的结束。

流是一个抽象的逻辑设备,它统一了各种文件的 I/O 接口,使得我们可以很简单地进行逻辑上的输入输出操作,而实际的实现则由 C 语言的文件系统自动地完成。C 语言本身并不支持文件的输入和输出,它没有定义相关操作语句的关键字。其输入输出操作都是由系统库函数来完成的,这样有利于系统开发和移植。根据其对文件的操作的两种不同方式:缓冲 I/O 和非缓冲 I/O,这些函数分成两组:一组叫“缓冲文件系统”,另一组叫“非缓冲文件系统”。

缓冲文件系统是由 ANSI 标准定义的。所谓缓冲文件系统是指在对文件进行操作时,系统将自动地在内存中为每一个正在使用的文件开辟一块区域作为缓冲区,文件的存取都通过缓冲区来进行。其方式是这样的:当程序向文件进行读操作时,先将数据读入读缓冲区,在缓冲区装满之后再向程序数据区传送数据,由程序各变量来接收;写操作时则是由程序将数据写入写缓冲区,缓冲区填满之后再写入文件。由此可以看出,缓冲区的大小决定了系统对文件的实际操作次数,这在很大程度上影响着程序的效率。缓冲区大小由系统确定,一般为 512 字节。缓冲文件系统通过缓冲区来支持流操作,在设计上可以对多种不同的设备如终端(键盘、显示器等)、磁盘驱动器和磁带等进行读写。

非缓冲文件系统并不为 ANSI 标准所支持,它源于 UNIX 操作系统对文件系统的设计,使用缓冲文件系统处理文本文件而使用非缓冲文件系统处理二进制文件。在非缓冲文件系统中,系统不会自动为文件读写开辟缓冲区,每个文件读写所需的临时存储数据的内存空间都由程序来设定和管理,程序员需要处理更多的细节。

ANSI 标准扩充了缓冲文件系统的功能,它也用来读写二进制文件,我们只重点介绍缓冲文件系统。

8.2 缓冲文件系统

在缓冲文件系统中,文件都是通过文件指针来进行操作的。文件指针是一个结构指针,相应结构在 stdio.h 中被定义成类型 FILE,它包含有关文件、缓冲区及文件操作类型等必需的信息:

```
typedef struct
{
    /* 文件、缓冲区及文件操作类型 */
}FILE;
```

因此,程序中对文件操作时,应首先定义文件指针变量,如:

```
FILE *fp;
```

这样我们就可以通过文件指针 fp 使用缓冲文件系统各函数来进行文件的读写。缓冲文件系统各函数的函数原型都包含在头文件 stdio.h 中,因此,用这些函数进行文件操作时,程序必须用 #include 编译预处理命令包含这个头文件。

8.2.1 文件的打开、关闭和文件结束测试

在能够进行读写操作之前必须先将文件打开,缓冲文件系统在这里主要进行三项工作:

- 指定准备进行访问的文件名字
- 指定文件的使用方式
- 指定一个文件指针,该指针指向文件操作所必需的信息

fopen()函数用来完成这些工作,其函数原型如下:

```
FILE *fopen(const char *filename, const char *mode);
```

该函数调用将打开一个文件,通常使用方式如下:

```
FILE *fp;
if((fp=fopen("myfile", "r"))==NULL)
{
    printf("文件打开失败!");
    exit(0);
}
```

filename 是一个字符串组成的有效文件名,允许带有路径名。mode 是说明文件打开方式的字符串,有效值定义如下(表 8.1):

表 8.1 文件的打开方式

文 本 文 件		二 进 制 文 件	
mode	含 义	mode	含 义
"r"	只读形式打开一个文本文件	"rb"	只读形式打开一个二进制文件
"w"	只写形式创建一个文本文件	"wb"	只写形式创建一个二进制文件
"a"	追加形式打开一个文本文件	"ab"	追加形式打开一个二进制文件
"r+"	读写形式打开一个文本文件	"rb+"	读写形式打开一个二进制文件
"w+"	读写形式创建一个文本文件	"wb+"	读写形式创建一个二进制文件
"a+"	读写形式打开一个文本文件	"ab+"	读写形式打开一个二进制文件

此表中各方式字符描述打开方式如下:

1. r: 用于以只读方式打开一个已存在文件,打开后只能从该文件中读取数据。
2. w: 用于以只写方式创建一个新文件,创建后只能向该文件中写入数据,若文件名指定的文件已存在,它的内容将被删去(刷新)。
3. a: 用于以追加方式打开一个已存在文件,打开后可以在文件尾部添加数据。
4. b: 用于表明打开或创建的文件是二进制文件,此时将一个二进制流与相应文件联系;缺省字符 b 则表明是文本文件,与文本流相联系。
5. +: 用于表明打开或创建的文件允许读写两项操作,我们也可称这种方式为更新方式,即文件位置指针不在文件尾时进行写操作将以覆盖方式写。

6. r 和 a 两种打开方式的差别在于文件被打开时,文件的位置指针不同,前者总是在文件首,而后者在写时是在文件尾,读("a+"或"ab+")时从文件首开始。

fopen()函数的返回值是一个 FILE 类型的文件指针,若打开文件成功,该指针将指向相应文件操作所必需的数据,因此我们必须先定义一个 FILE 型指针变量来接收该信息。若文件打

开操作失败,它将返回一个空指针 NULL 值(NULL 在 `stdio.h` 中被定义为零值)。在这里使用 `if` 语句是文件打开语句的常规做法,保证后面的文件读写是对一个有效打开的流进行的,这样程序才能正确运行。

在程序运行之初,系统将自动打开三个标准流:标准输入、标准输出和标准出错输出,它们均与终端相联系。系统还定义了三个文件指针 `stdin`、`stdout` 和 `stderr`,它们分别指向这三个标准设备。如果程序在文件读写操作时使用这些指针,将是对终端设备的读写操作。系统还定义了标准打印机和标准辅助设备,分别以指针 `stdprn` 和 `stdaux` 指向它们。

在文件不再被使用时,应尽早关闭它。主要基于两点理由:一是节省资源,被打开的文件总会耗费一定的系统资源,如具体系统允许同时打开的文件数和缓冲区数都是有一定限数的;二是安全方面的考虑,防止误用或丢失信息。

`fclose()` 函数用于关闭一个用 `fopen()` 打开的流。它完成下面的工作:

1. 将缓冲区中的数据写入到文件中去。系统只在缓冲区满时才自动将其内容写入文件,因此程序最后几次“写文件”的内容可能还在缓冲区,没有真正写入文件中。
2. 系统级上的文件关闭操作,释放与该文件的操作相关的系统资源。

`fclose()` 函数的函数原型如下:

```
int fclose(FILE *fp);
```

该函数的调用将关闭一个文件指针 `fp` 所指出的文件及相关资源,`fp` 是在 `fopen()` 函数调用时被赋值的文件指针。`fopen()` 和 `fclose()` 分别完成缓冲文件系统对某文件进行操作时的开始和结尾工作,而文件指针将贯穿始终。当 `fclose()` 正常关闭时将返回 0 值,否则返回 EOF。

EOF 是文件结束标志,被定义为整型常量 -1。在进行文件读写操作时,常常需要对文件是否已到结尾位置进行判断。在文本流中使用 EOF 来作为文件结束标志是十分合适的:文本流将文件中数据映射(转换)为一个字符序列,该序列均以 ASCII 码表示,不会出现 -1 这个值;但在二进制流中,其字节序列与文件中数据一一对应,没有转换过程,因而每个字节的取值可以是任意的,包括 -1 这个值,EOF 将不能对文件结束作出正确的判断,此时就需要对文件是否真正结束作出测试,这项工作是由函数 `feof()` 来进行的,它的函数原型如下:

```
int feof(FILE *fp);
```

该函数利用文件指针 `fp` 对相应文件进行检查,并判断相应的文件操作是否已到了文件的结尾位置。在文件正常结束时返回非 0 值,否则返回 0。它不仅适用于对按二进制方式打开的文件,也同样可以用于文本流。

本节介绍的函数 `fopen()` 和 `fclose()` 是程序中使用缓冲文件系统进行读写操作所必须调用的函数,`feof()` 函数也是很常用的,我们将在后面结合文件的读写操作给出它们的示例。

8.2.2 文件的读写

一旦正常打开了一个文件,我们就可以对它进行读或写的操作。在缓冲文件系统中,根据文件的打开方式,其操作可以是建立在二进制流或文本流之上的,虽然它们都是字节(字符)序列,但二进制流与文件内容一一对应,文本流与文件内容之间存在着互相转换的问题,并不完全一致。针对操作数据的不同,文件读写的函数分为四类:字符输入输出函数和字符串输入输出函数,以及格式化输入输出函数和数据块读写函数。

1. 字符输入输出

(1) fgetc 和 fputc 函数 (getc 和 putc 函数)

函数原型: int fgetc(FILE *fp);

int fputc(int ch, FILE *fp);

功能: 从 fp 指定的流中读取一个字符

写一个字符到 fp 指定的流中

返回值: 操作成功时, 两函数均返回被操作的字符, 否则均返回 EOF。

[例 8.1] 将文件 file1 复制到文件 file2。

```

#include "stdio.h"
main()
{
    int c;
    FILE *fp1, *fp2;
    if((fp1=fopen("file1", "r"))==NULL)
    {
        printf("不能打开文件 file1! \n");
        exit(1);
    }
    if((fp2=fopen("file2", "w"))==NULL)
    {
        printf("不能打开文件 file2! \n");
        exit(1);
    }
    while((c=fgetc(fp1))!=EOF) fputc(c,fp2);
    fclose(fp1);
    fclose(fp2);
}

```

程序首先定义了一个变量 c 用以在两个文本流之间传送字符, 两次 fopen() 函数调用建立了这两个流, 并分别只读地打开文件 file1 和创建名为 file2 的文件。其中用到的两个文件指针也在一开始定义为 FILE 结构型指针。复制工作在一条 while 语句中完成; 在 while 语句的条件表达式中, 使用 fgetc() 函数读入一个字符并判断该操作是否成功; 循环体中通过函数 fputc() 的调用写出字符, 这两个函数对流进行操作, 而具体对文件的读和写由系统自动完成。复制完成后用 fclose() 函数关闭这两个文件结束程序。

在[例 8.1]中, 我们也可以用二进制流的方式来打开文件, 相应地在 fopen() 函数中打开及创建文件时, 字符串应改为 "rb" 和 "wb", 同时 while 语句应改为:

```
while(! feof(fp1)) fputc(fgetc(fp1),fp2);
```

函数 getc() 和 putc() 的功能和调用格式分别与函数 fgetc() 和 fputc() 完全一样, 通常我们根本不必理会它们之间细微差别。C 语言就是这样, 与其他高级语言相比, C 系统是一个小内核的系统, 语言本身并不包含与操作系统等外部环境有关的内容, 而这些工作都交给函数去做, 这样使得 C 语言极具灵活性并易于功能扩充, 同时也导致了相同功能的函数的若干不同版本出现, 一般情况我们都可以不加区分地使用它们。

(2) getchar 和 putchar 函数

它们是 `fgetc` 和 `fputc` 两个函数派生出来的宏。如前所述,终端设备(键盘、显示器)也是文件,并在程序运行时自动打开,相应的文件指针为 `stdin` 和 `stdout`。在 Turbo C 中如下定义 `getchar` 和 `putchar`;

```
#define getchar() getc(stdin)
#define putchar(c) putc((c),stdout)
```

可见它们实际是两个宏,就使用的角度而言,我们不需要关心系统是如何实现某项功能的,因而也没有必要区分系统中的宏和函数这两个概念,所以在这里统一称之为函数。事实上 `feof()` 也是被定义为宏的,但我们完全可以把它们当作函数来使用。

终端的读和写在程序中经常使用。键盘和显示器(文本方式下)都是字符设备,相应 `stdin` 和 `stdout` 均指文本流,因此, `getchar` 和 `putchar` 是对一个文本流进行操作的。当我们利用 `getchar()` 读入字符时,输入字符首先被送入缓冲区,只有在按下回车键结束一行时,缓冲区的内容才开始被 `getchar()` 函数读取,回车键键入的字符被读入为换行符 `'\n'`;

(3) `ungetc` 函数

函数原型: `int ungetc(int ch, FILE *fp);`

功能:将刚读入的变量 `ch` 中的字符退回到流中。

返回值:成功时返回所退回的字符的 ASCII 值,否则返回 `EOF`。

[例 8.2] 下面函数从文件中连续读取数字字符,遇到非数字字符时退回这个字符到流中以备后用,然后结束函数调用。

```
void read _digit(FILE *fp, char *str)
{
    char c;
    while((c=getc(fp))>'0'&& c<'9') *(str++)=c;
    ungetc(c,fp);
}
```

2. 字符串输入输出

`fgets` 和 `fputs` 函数

函数原型: `char *fgets(char *str, int length, FILE *fp);`

`int fputs(char *str, FILE *fp);`

功能: `fgets()` 函数从文件指针 `fp` 指定的流中读入一个长为 `length` 的字符串(放在 `str` 中); `fput()` 函数写字符串 `str` 到 `fp` 指定的流中。

返回值: `fgets()` 函数返回读入的字符串首地址,读到文件尾或出错时返回一个空指针 (`NULL`); `fputs()` 函数在写串成功时返回非负整数,否则返回 `EOF`。

[例 8.3] 用 `getc()` 和 `putc()` 函数实现函数 `fgets()` 和 `fputs()`。

```
#include "stdio.h"
char *fgets(char *str, int length, FILE *fp)
{
    int c;
    char *s;
    s=str;
    while(--length>0&&(c=getc(fp))!=EOF)
```



```

        if(( * s++ = c) == '\n') break;
        * s = '\0';
        return ((c == EOF && s == str)?NULL:str);
    }

    fputs(char * str, FILE * fp)
    {
        int c;
        while(c = * str++) putc(c, fp);
    }

```

getc()函数是从标准输入流中输入字符串的,它与 fgets()从键盘输入(stdin 作为文件指针)有所区别:getc()把读入的换行符替换为'\0',而 fgets()则保留换行符,它在读入指定长度的字符后补'\0',使之成为字符串。

putc()函数则是向标准输出流输出字符串,它和 fputs()函数向标准输出流(stdout 作为文件指针)输出时也有不同:putc()将要输出的字符串结束符('\0')转换成换行符放入标准输出流中,而 fputs()只是将字符串结束符('\0')丢弃掉,其他内容原样放入标准输出流中。

3. 格式化输入输出

(1) fprintf 和 fscanf 函数

函数原型: int fprintf(FILE * fp, const char * format, ...);

int fscanf(FILE * fp, const char * format, ...);

功能: fprintf()/fscanf()函数分别以格式控制串(format)所指定的格式,向/从 fp 所指定的流输出/读入数据,数据项被列写在格式控制串后的参数表中。

返回值: fprintf()返回实际被写的字符个数,若出错则返回一个负数; fscanf()返回实际被赋值的参数个数,返回 EOF 值则表示试图去读取超过文件尾端的部分。

fprintf()与 fscanf()函数的使用十分类似于 printf()和 scanf()函数,后者专门用于标准输入输出流的操作,而前者主要用于对磁盘文件的格式化读写。

利用 fprintf()和 fscanf()函数可以方便地将程序的(中间)结果按指定格式存放在磁盘文件中,特别对于数值数据,可以减少在程序中对其进行字符化的处理。

[例8.4] 下面程序生成文件 list.dat,以“单价 数量”格式存放某商场商品数据,其总金额存放在 list.dat 文件的最后一行。

```

#include "stdio.h"
main()
{
    FILE * fp;
    double total=0;
    float price, p[3]={12.3, 45.6, 78.9};
    int i, num, n[3]={10, 20, 30};
    fp=fopen("list.dat", "w");
    for(i=0; i<3; i++)
        fprintf(fp, "%4.2f %d\n", p[i], n[i]);
    fclose(fp);
}

```

```

        fp=fopen("list.dat","a+");
        while(!feof(fp))
        {
            fscanf(fp,"%f %d\n",&price,&num);
            total+=price * num;
        }
        fprintf(fp,"%s %12.3f","总金额:",total);
        fclose(fp);
    }

```

程序运行结束后将生成一个名为 list.dat 的文件,其内容如下:

```

12.30 10
45.60 20
78.90 30
总金额:      3402.000

```

另外还有两个与 fprintf() 和 fscanf() 十分相似的函数,它们是 sprintf() 和 sscanf() 函数。

(2) sprintf 和 sscanf 函数

函数原型: int sprintf(char *str, const char *format, ...);

int sscanf(char *str, const char *format, ...);

功能: sprintf()/sscanf() 函数分别以格式控制串(format)所指定的格式,向/从 str 所指定的字符串输出/读入数据,数据项被列写在格式控制串后的参数表中。

返回值: sprintf() 返回实际写入数组的字符个数; fscanf() 返回实际被赋值的字段数。

例如:

```
sprintf(str,"%s %d %c","one",2,'3');
```

语句执行后字符串 str 将为 "one 2 3"; 而执行以下语句:

```
sscanf("hi 1 2.3","%s%d%f",s,&i,%f);
```

后字符串 s 为 "hi", 整型变量 i 的值为 1, 浮点变量 f 的值为 2.3。

4. 数据块读写

fread 和 fwrite 函数

函数原型: int fread(void *ptr, int size, int count, FILE *fp);

int fwrite(void *ptr, int size, int count, FILE *fp);

功能: fread() 函数从 fp 指向的流中读取 count(字段数)个字段,每个字段为 size(字段长度)个字符长,并把它们放到指针 ptr 指向的字符数组(缓冲区)中; fwrite() 函数从指针 ptr 指向的字符数组中,把 count 个字段写到 fp 指向的流中去,每个字段为 size 个字符长。

返回值: fread()/fwrite() 函数返回实际已读取/写入的字段个数,如果实际的个数少于所要求的(count)个数,则操作失败。

[例8.5] 数据块读写示例。

```

#include "stdio.h"
main()
{
    FILE *fp;

```



```

static float a[2][2]={0.1,2.3,4.5,6.7};
float b[4];
int i;
fp=fopen("test","wb");
fwrite(a,sizeof(float),4,fp);
fclose(fp);
fp=fopen("test","rb");
fread(b+2,sizeof(float),2,fp);
fread(b,sizeof(float),2,fp);
for(i=0;i<4;i++)
    printf("%f%c",i%2==1?'\\n':' ',b[i]);
fclose(fp);
}

```

运行结果:

```

4.500000 6.700000
0.100000 2.300000

```

8.2.3 文件的定位

在C语言中,文件内部没有记录、数据项等结构,在缓冲型文件系统中它是按流方式读写的,在文件的读写过程中,文件位置指示器顺序地以字符(节)为单位移动,这样实现文件的顺序读写。要实现文件的随机读取,就需要对文件位置指示器进行设置,这可以通过以下函数来实现。

1. fseek 函数

函数原型: `int fseek(FILE *fp, long int offset, int origin);`

功能: 将 `fp` 指向的文件的文件指针移动到由起点(`origin` 指定)偏移 `offset` 个字节的位置处。

返回值: 正确执行时返回0,否则为非零值。

`origin` 指定的起点是表8.2所示的几个宏之一:

表8.2 `fseek()`函数的形参 `origin` 取值表

origin(起点)	宏名字	常量值
文件开头	SEEK_SET	0
当前位置	SEEK_CUR	1
文件尾端	SEEK_END	2

这些宏在头文件 `stdio.h` 中被定义为整型常量, `SEEK_SET` 为0,代表文件开头位置; `SEEK_CUR` 为1,代表当前位置; `SEEK_END` 为2,代表文件尾端位置。

偏移量 `n` 被定义为长整型,实参为常量时书写应用 `nL` 表示,其值表示移动的字节数,即相对于 `origin` 指定的起点位置的字节数目。

由于使用文本流读写文件时存在某些字符的翻译转换,如流中的单个字符 `'\\n'` 对应文件

中的回车换行两个字符,故在使用 `fseek()` 函数时,建议用二进制流打开文件,避免应用文本流而可能造成的位置错误。

[例8.6] 下面程序以十六进制数和字符两种方式同时显示指定扇区的文件内容。文件为 `disp.c`。

```
#include "stdio.h"
#include "ctype.h"
#define SIZE 128
char buf[SIZE];
void display(int);
main(int argc, char * argv[])
{
    FILE * fp;
    long int sector, num;
    fp = fopen(argv[1], "rb");
    do
    {
        printf("请输入扇区号:");
        scanf("%ld", &sector);
        if(sector >= 0)
        {
            fseek(fp, sector * SIZE, SEEK_SET);
            if((num = fread(buf, 1, SIZE, fp)) != SIZE)
                printf("到达了文件尾部!\n");
        }
        display(num);
    } while(sector >= 0);
    fclose(fp);
}
void display(int num)
{
    int i, j;
    for(i = 0; i < num / 16; i++)
    {
        for(j = 0; j < 16; j++) printf(" %02X", buf[i * 16 + j]);
        printf(" | ");
        for(j = 0; j < 16; j++)
            if(isprint(buf[i * 16 + j])) printf(" %c", buf[i * 16 + j]);
        else printf(". ");
        printf("\n");
    }
}
```

}

在DOS命令行上输入 disp disp.c 回车,然后键入1回车,屏幕显示:

C:>disp disp.c

请输入扇区号: 1

```
61 72 67 63 2C 63 68 61 72 20 2A 61 72 67 76 5B | argc,char * argv[
5D 29 0D 0A 20 20 20 20 7B 0D 0A 20 20 20 20 20 | ])..      {..
20 46 49 4C 45 20 2A 66 70 3B 0D 0A 20 20 20 20 | FILE * fp;..
20 20 6C 6F 6E 67 20 69 6E 74 20 73 65 63 74 6F | long int secto
72 2C 6E 75 6D 3B 0D 0A 20 20 20 20 20 20 66 70 | r,num;..      fp
3D 66 6F 70 65 6E 28 61 72 67 76 5B 31 5D 2C 22 | =fopen(argv[1],"
72 62 22 29 3B 0D 0A 20 20 20 20 20 20 64 6F 0D | rb");..      do.
0A 20 20 20 20 20 20 7B 0D 0A 09 70 72 69 6E 74 | .      {... print
```

程序中使用了系统库函数 isprint(),它用于判断它的参数是否为一个可打印字符,其函数原型在头文件 ctype.h 中。fseek()函数定位指定扇区,然后利用 fread()函数读入该扇区的内容。当输入扇区号为负数时,结束程序的运行。

2. rewind 函数

函数原型: int rewind(FILE *fp);

功能: rewind()函数将文件指针移到 fp 所指定的文件的起始位置。

返回值: 成功时返回0,否则返回非零值。

[例8.7] 使用 rewind()函数改写[例8.5]。

```
#include "stdio.h"
main()
{
    FILE *fp;
    static float a[2][2]={0.1,2.3,4.5,6.7};
    float b[4];
    int i;
    fp=fopen("test","wb+");
    fwrite(a,sizeof(float),4,fp);
    fread(b+2,sizeof(float),2,fp);
    fread(b,sizeof(float),2,fp);
    for(i=0;i<4;i++)
        printf("%f%c",i%2==1?'\\n':' ',b[i]);
    fclose(fp);
}
```

3. ftell 函数

函数原型: long int ftell(FILE *fp);

功能: 指出文件指针的当前位置。

返回值: 成功时返回文件指针相对于文件起点的位移的长整型字节数,失败时返回-1L。

可以利用 ftell()函数来记录当前文件指针的位置,以备它用。

8.2.4 出错的处理

对文件的操作可能会因程序及程序之外的原因导致出错,我们可以用以下两个函数来进行处理。

1. ferror 函数

函数原型: `int ferror(FILE *fp);`

功能:检测 `fp` 指定的流中的文件错误。

返回值:返回值为0时,表示没有出现错误;而非零值表示有错。

在调用各种文件输入输出函数时,如果出现了错误,我们均可以使用 `ferror()` 函数来进行检查。当与 `fp` 相关联的出错标记给出后,将保持到该文件的下一次读写操作之前,或至调用了函数 `rewind()` 或 `clearerr()` 为止。

`ferror()` 实际上是一个宏,它被定义为 `FILE` 型指针 `fp` 指向的相应流的出错标记。每次文件的输入输出操作后,其成功与否都将记录在出错标记中。初始时在执行 `fopen()` 函数调用后,出错标记被自动置为0。

2. clearerr 函数

函数原型: `void clearerr(FILE *fp);`

功能:它把 `fp` 指向的文件出错标记重新置为0,同时文件的结束指示器也被重新设置。

8.3 非缓冲文件系统

非缓冲文件系统不是 ANSI C 所支持的,这里只作简要的基本介绍,实际使用中应查阅 C 手册以了解更多的内容。不同于缓冲文件系统的输入输出,系统不为文件准备缓冲区,也就没有结构 `FILE` 及其指针。在程序存取文件时和文件联系的是通过一个称为文件描述字的正整数来实现的,函数通过这个文件描述字来读写文件以及外部设备。非缓冲文件系统函数原型都包含在 `io.h` 中。

标准设备的文件描述字是固定的,由系统自动分配,它们的值如下(表8.3):

表8.3 标准设备的文件描述字

标准设备	文件描述字
标准输入	0
标准输出	1
标准错误输出	2

1. 文件的建立、打开和关闭

在非缓冲文件系统中,文件也需要打开了才能使用,使用完后也应该关闭。

(1) creat()

函数原型: `int creat(const char *filename, int mode);`

功能:以一定的读写模式创建一个字符串 `filename` 指定文件名的新文件。

返回值:创建成功时返回一个正整数作为与该文件相联系的文件描述字,否则返回-1。

函数参数 mode 决定文件访问模式,如宏(在文件 sys/stat.h 中定义)S_IWRITE 和 S_IREAD 分别用于文件的只写方式和只读方式创建,而 S_IWRITE|S_IREAD 用于读写方式。

在调用 creat()函数时,若指定文件已存在,则原内容会被抹去,除非该文件有写保护。

(2)open()

函数原型: `int open(const char * filename, int access, [unsigned mode]);`

功能:以一定的读写模式打开一个字符串 filename 指定文件名的文件。

返回值:打开成功时返回一个正整数作为与该文件相联系的文件描述字,否则返回-1。

函数参数 access 指定文件的访问方式,基本的取值有宏:

O_RDONLY 打开文件只读

O_WRONLY 打开文件只写

O_RDWR 打开文件读写

选定上述其中一个值后,还可以用或(|)方式与下述值的一个或多个一起使用:

O_APPEND 在每次写操作前把文件指针置于文件尾端

O_TRUNC 如果文件存在,将其长度置为零,但保持其原属性

O_CREAT 如果文件不存在,生成一个以 mode 值为属性的文件

O_BINARY 打开一个二进制文件

O_TEXT 打开一个文本文件

仅在使用 O_CREAT 时需要函数参数 mode,其取值同 creat()的函数参数 mode。

以上宏均在头文件 fcntl.h 中定义。

(3)close()

函数原型: `int close(int fd);`

功能:关闭与文件描述字 fd 相联结的文件。

返回值:如果调用成功将返回0值,否则返回-1。

(4)eof()

函数原型: `int eof(int fd);`

功能:测试与文件描述字 fd 相联结的文件指针是否已达到文件尾端。

返回值:如果到达了文件尾则返回1,否则返回0,出错时返回-1。

2. 文件的读写

一个文件在打开后就可以进行读或写的操作,这通以以下两个函数来完成。

函数原型: `int read(int fd, void * buf, unsigned size);`

`int write(int fd, void * buf, unsigned size);`

功能:read()/write()函数从/向与 fd 相联结的文件读取/写入 size 个字节的字符,这些字符写到/源于指针 buf 所指向的内存区域。

返回值:read()/write()函数返回实际读取/写入的字节数,出现错误时返回-1。

[例8.8] 使用非缓冲文件系统经进行文件复制。

```
#include <io.h>
```

```
#include <fcntl.h>
```

```
#include <sys\stat.h>
```



```

main()
{
    int fdr,fdw,n;
    char buf[256];
    fdr=open("123",O_RDONLY);
    fdw=creat("abc",S_IWRITE);
    while(!eof(fdr))
    {
        n=read(fdr,buf,256);
        write(fdw,buf,n);
    }
    close(fdr);
    close(fdw);
}

```

3. 随机访问

与缓冲文件系统的 `fseek()` 函数类似,在非缓冲文件系统中使用 `lseek()` 函数实现文件的随机访问。

函数原型: `long lseek(int fd, long offset, int origin);`

功能:将与 `fd` 相联结的文件的文件指针移动到由起点(`origin` 指定)偏移 `offset` 个字节的位置处。

返回值:调用成功时返回从文件开始到当前位置的字节数,失败时返回 `-1L`。

函数参数 `origin` 的取值同 `fseek()` 的函数参数 `origin` (参见表 8.2,相关的宏在 `io.h` 中也有定义)。

[例 8.9] 用非缓冲文件系统改写[例 8.6]。

```

#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <ctype.h>
#define SIZE 128
char buf[SIZE];
void display(int);
main(int argc, char *argv[])
{
    long int sector, num;
    int fd;
    fd=open(argv[1], O_RDONLY|O_BINARY);
    do
    {
        printf("请输入扇区号:");
        scanf("%ld", &sector);
    }
}

```

```

        if(sector >= 0)
        {
            lseek(fd, sector * SIZE, SEEK_SET);
            if((num = read(fd, buf, SIZE)) != SIZE) printf("到达了文件尾部!\n");
        }
        display(num);
    } while(sector >= 0);
    close(fd);
}

void display(int num)
{
    int i, j;
    for(i = 0; i < num / 16; i++)
    {
        for(j = 0; j < 16; j++) printf(" %02X", buf[i * 16 + j]);
        printf(" | ");
        for(j = 0; j < 16; j++)
            if(isprint(buf[i * 16 + j])) printf(" %c", buf[i * 16 + j]);
        else printf(". ");
        printf("\n");
    }
}

```

习 题

一、选择题

8.1 函数【1】用于文件的格式输入。

- 【1】 A) read() B) fread() C) scanf() D) fscanf()

8.2 以下叙述正确的是【2】。

- 【2】 A) 函数 fprintf() 不能将数据输出到标准输出流
 B) 程序中必须明确地用 fopen() 打开标准输入输出流
 C) 查找随机存取文件中的指定记录不必从头逐条查找
 D) 函数 rewind() 可以定位文件的任意指定位置

8.3 函数 fseek() 用于【3】。

- 【3】 A) 查找文件 B) 定位文件指针
 C) 查找字符 D) 定位文件开始位置

8.4 下面函数调用正确的是【4】。

- 【4】 A) fopen('file.dat', 'r'); B) fprintf("%ld", data, fp);
 C) fclose("file.dat"); D) fread (&i, sizeof(int), 1, fp);

8.5 若 fp 为已正确打开的文件指针, i 为 long 型变量, 以下语句的输出结果是【5】。

```
fseek( fp, 0, SEEK _END );
i=ftell( fp );
printf( "i=%ld\n", i );
```

【5】 A)-1

B)fp 所指文件的长度, 以字节为单位

C)0

D)2

二、填空题

8.6 下面程序将文件内容全部输出到屏幕上, 其文件名从命令行参数中获得。

```
#include <stdio.h>
main(argc,argv)
int 【1】;
char 【2】;
{ FILE *fp;
  char ch;
  if(argc<2)
  {
    printf("error;no file name!\n");
    exit(1);
  }
  if((fp=fopen( 【3】 ,"r"))==NULL)
  {
    printf("Can't open file! \n");
    exit(2);
  }
  while ((ch=getc(fp))!= 【4】 ) printf("%c",ch);
  fclose( 【5】 );
}
```

三、改错题

8.7 下面程序建立一个随机存取文件, 请改正程序中的错误。

```
#include <stdio.h>
struct Data{
  int account;
  char name[8];
  int age;
};
main()
{
  int i;
  struct Data blank = {0, "", 0};
  FILE fp;
```

```
if (fp=fopen("newfile","rw")==NULL)
    printf("打开文件错误! \n");
else
{
    for(i=0;i<=100;i++)
        fwrite(blank,sizeof(struct Data),1,fp);
    fclose(fp);
}
```

四、程序设计题

- 8.8 编写程序输出源程序,在打印时每行均加上行号。
- 8.9 编写程序,在指定文件中查找指定的字符串所在的行和列,找不到时返回-1。
- 8.10 编写程序,求某数据文件中的所有数据的平均值、最大值和第二大的值。
- 8.11 编写程序,其功能为复制文件,但将原文件每段首字母转换为大写。
- 8.12 编写一个简单的通讯录,指定人名则输出电话号码。

※ 第 9 章

实用程序设计初步

学习程序设计是为了设计应用程序,用来解决实际生活中可能遇到的各种问题。实际问题是多种多样的,往往需要编程者掌握相当程度的相关专业知识和丰富的程序设计的知识和经验。一个完整实用的应用程序应具有良好的用户界面及清晰直观的输出结果,这通常是利用图形程序设计实现的。另外,为了设计好的应用程序,还需要经常与硬件打交道,以实现计算机硬、软件资源的充分利用,因此,我们须了解中断调用程序设计的实现。本章就这两个问题进行讨论,作为学习实用程序设计的入门。

C 语言是一种小内核的程序语言,与系统相关的功能是通过函数实现的,本章介绍的图形处理和中断调用的程序设计都是利用系统库函数来进行。作为入门,也由于这部分内容涉及到较多的软硬件知识,这里仅介绍与此相关的基础知识、编程规范和简单例子,实际程序设计时可能还需要查阅相关手册。

9.1 图形处理

Turbo C 提供了非常丰富的图形函数以实现图形程序设计,所有图形函数的原型均在 graphics.h 中,本节主要介绍图形模式的初始化、独立图形程序的建立、基本图形功能、图形窗口以及图形模式下的文本输出函数等。

使用图形函数时要确保有显示器图形驱动程序“*.BGI”,同时将集成开发环境 Options/Linker 中的 Graphics lib 选为 on,只有这样才能保证正确使用图形函数。

9.1.1 图形模式的初始化

不同的显示器适配器有不同的图形分辨率,即使是同一显示器适配器,在不同模式下也有不同分辨率。因此,在屏幕作图之前,必须根据显示器适配器种类将显示器设置成为某种图形模式。在未设置图形模式之前,微机系统默认屏幕为文本模式(80 列、25 行的字符模式),此时所有图形函数均不能工作。设置屏幕为图形模式可用下列图形初始化函数:

```
void far initgraph(int far *gdriver, int far *gmode, char *path);
```

其中 gdriver 和 gmode 分别代表图形驱动器和模式,path 是指图形驱动程序所在的目录路径。有关图形驱动器、图形模式的符号常数及对应的分辨率见表 9.1。

图形驱动程序由 C 系统出版商提供,文件扩展名为 BGI。根据不同的图形适配器有不同的图形驱动程序。例如对于 EGA、VGA 图形适配器就调用驱动程序 EGAVGA.BGI。

表 9.1 图形驱动器、模式的符号常数及数值

图形驱动器(gdriver)		图形模式(gmode)		分辨率	调色板
符号常数	数值	符号常数	数值		
DETECT	0	用于自动检查			
CGA	1	CGAC0	0	320×200	C0
		CGAC1	1	320×200	C1
		CGAC2	2	320×200	C2
		CGAC3	3	320×200	C3
		CGAHI	4	640×200	2 色
MCGA	2	MCGAC0	0	320×200	C0
		MCGAC1	1	320×200	C1
		MCGAC2	2	320×200	C2
		MCGAC3	3	320×200	C3
		MCGACMED	4	640×200	2 色
		MCGACHI	5	640×480	2 色
EGA	3	EGALO	0	640×200	16 色
		EGAHI	1	640×350	16 色
EGA64	4	EGA64LO	0	640×200	16 色
		EGA64HI	1	640×350	4 色
EGAMON	5	EGAMONHI	0	640×350	2 色
IBM8514	6	IBM8514LO	0	640×480	256 色
		IBM8514HI	1	1024×768	256 色
HERC	7	HERCMONHI	0	720×348	2 色
ATT400	8	ATT400C0	0	320×200	C0
		ATT400C1	1	320×200	C1
		ATT400C2	2	320×200	C2
		ATT400C3	3	320×200	C3
		ATT400MED	4	640×200	2 色
		ATT400CHI	5	640×400	2 色
VGA	9	VGALO	0	640×200	16 色
		VGAMED	1	640×350	16 色
		VGAHI	2	640×480	16 色
PC3270	10	PC3270HI	0	720×350	2 色

[例 9.1] 使用图形初始化函数设置 VGA 高分辨率图形模式。

```
#include <graphics.h>
main()
{
    int gdriver, gmode;
    gdriver=VGA;
    gmode=VGAHI;
    initgraph(&gdriver, &gmode, "c:\\tc");
    ...
}
```

有时编程者并不知道所用的图形显示器适配器种类,或者需要将编写的程序用于不同图形驱动器,Turbo C 提供了一个自动检测显示器硬件的函数,其调用格式为:

```
void far detectgraph(int *gdriver, *gmode);
```

其中 gdriver 和 gmode 的意义与上面相同。

[例 9.2] 硬件测试后进行图形初始化。

```
#include <graphics.h>
main()
{
    int gdriver, gmode;
    detectgraph(&gdriver, &gmode);           /* 自动测试硬件 */
    initgraph(&gdriver, &gmode, "c:\\tc");    /* 根据测试结果初始化图形 */
    ...
}
```

上例程序中先对图形显示器自动检测,然后再用图形初始化函数进行初始化设置,Turbo C 提供了一种更简单的方法,上例可改为如下形式:

```
#include <graphics.h>
int main()
{
    int gdriver=DETECT, gmode;
    initgraph(&gdriver, &gmode, "c:\\tc");
    ...
}
```

另外,Turbo C 提供了退出图形状态的函数 closegraph(),用于从图形方式回到屏幕的字符显示方式,其调用格式为:

```
void far closegraph(void);
```

调用该函数后可退出图形状态而进入文本方式(Turbo C 默认方式),并释放用于保存图形驱动程序和字体的系统内存。

9.1.2 独立图形运行程序的建立

Turbo C 对于用 `initgraph()` 函数直接进行的图形初始化程序,在编译和连接时并没有将相应的驱动程序(*.BGI)装入到执行程序,当程序进行到 `initgraph()` 语句时,再从该函数中第三个形式参数 `char * path` 所指定的路径中去找相应的驱动程序。若没有找到,则在 `C:\TC` 目录中去找,如果 `C:\TC` 中仍没有相应文件或 `TC` 目录不存在,将会出现错误:

BGI Error: Graphics not initialized (use 'initgraph')

因此为了使用方便,应该建立一个不需要驱动程序就能独立运行的可执行图形程序,Turbo C 中规定用下述步骤(这里以 EGA、VGA 显示器为例)来建立它:

1. 在 `C:\TC` 子目录下输入命令: `BGI OBJ EGAVGA`

此命令将驱动程序 `EGAVGA.BGI` 转换成 `EGAVGA.OBJ` 的目标文件。此时屏幕将提示驱动程序转换为函数 `EGAVGA _driver` 以供使用。

2. 在 `C:\TC` 子目录下输入命令: `TLIB LIB\GRAPHICS.LIB+EGAVGA`

此命令的意思是将 `EGAVGA.OBJ` 的目标模块装到 `GRAPHICS.LIB` 库文件中。

3. 在程序中 `initgraph()` 函数调用之前加上一句:

```
registerbgidriver(EGAVGA _driver);
```

该函数告诉连接程序在连接时把 `EGAVGA` 的驱动程序装入到用户的执行程序中。其函数原型如下:

```
int registerbgidriver(void (* driver)(void));
```

经过上面处理,编译连接后的执行程序可在任何目录或其他兼容机上运行。假设已作了前两个步骤,若再在[例 9.2]中加 `registerbgidriver()` 函数,则变成如下形式:

```
#include <stdio.h>
#include <graphics.h>
main()
{
    int gdriver=DETECT,gmode;
    registerbgidriver(EGAVGA _driver); /* 建立独立图形运行程序 */
    initgraph(&gdriver, &gmode,"c:\\tc");
    ...
}
```

上例编译连接后产生的执行程序可独立运行。

如不初始化成 EGA 或 VGA 分辨率,而想初始化为 CGA 分辨率,则只需将上述步骤中有 `EGAVGA` 的地方用 `CGA` 代替即可。

9.1.3 屏幕颜色的设置和清屏函数

图形模式下屏幕的颜色分为背景色和前景色。分别用下面两个函数进行设置:

设置背景色: `void far setbkcolor(int color);`

设置作图色: `void far setcolor(int color);`

其中 color 为图形方式下颜色的规定数值,对 EGA 或 VGA 显示器适配器,有关颜色的符号常数及数值见表 9.2 所示。

表 9.2 有关屏幕颜色的符号常数表

符号常数	数值	含义	符号常数	数值	含义
BLACK	0	黑色	DARKGREY	8	深灰
BLUE	1	蓝色	LIGHTBLUE	9	深蓝
GREEN	2	绿色	LIGHTGREEN	10	淡绿
CYAN	3	青色	LIGHTCYAN	11	淡青
RED	4	红色	LIGHTRED	12	淡红
MAGENTA	5	洋红	LIGHTMAGENTA	13	淡洋红
BROWN	6	棕色	YELLOW	14	黄色
LIGHTGRAY	7	淡灰	WHITE	15	白色

对于 CGA 适配器,背景色可以为表 9.2 中 16 种颜色的一种,但前景色依赖于不同的调色板。CGA 共有四种调色板,每种调色板上有四种颜色可供选择,如表 9.3 所示。

表 9.3 CGA 调色板与颜色值表

调色板		颜色值			
符号常数	数值	0	1	2	3
C0	0	背景色	绿	红	黄
C1	1	背景色	青	洋红	白
C2	2	背景色	淡绿	淡红	黄
C3	3	背景色	淡青	淡洋红	白

清除图形屏幕内容使用清屏函数,其调用格式如下:

```
void far cleardevice(void);
```

[例 9.3] 颜色设置、清屏函数的使用。

```
#include <stdio.h>
#include <graphics.h>
int main()
{
    int gdriver, gmode, i;
    gdriver=DETECT;
    registerbgidriver(EGAVGA _DRIVER); /* 建立独立图形运行程序 */
    initgraph(&gdriver, &gmode, ""); /* 图形初始化 */
    setbkcolor(0); /* 设置图形背景 */
    cleardevice();
    for(i=0; i<=15; i++)
    {
        setcolor(i); /* 设置不同作图色 */
        circle(320, 240, 20+i*10); /* 画半径不同的圆 */
        delay(100); /* 延迟 100 毫秒 */
    }
    for(i=0; i<=15; i++)
```

```

{
    setbkcolor(i);                /* 设置不同背景色 */
    cleardevice();
    circle(320, 240, 20+i*10);
    delay(100);
}
closegraph();
}

```

另外, Turbo C 也提供了几个获得现行颜色设置情况的函数:

```

int far getbkcolor(void);        返回现行背景颜色值
int far getcolor(void);          返回现行作图颜色值
int far getmaxcolor(void);       返回最高可用的颜色值

```

9.1.4 基本图形函数

基本图形函数包括画点、线函数以及其他一些基本图形的函数。

1. 画点

(1) 画点函数

```
void far putpixel(int x, int y, int color);
```

该函数表示在指定坐标(x,y)上画一个按 color 值所指定颜色(见表 9.2)的点。

在图形模式下,坐标是按像素点来定义的。对 VGA 适配器,它的最高分辨率为 640×480 , 其中 640 为整个屏幕从左到右所有像素点的个数,480 为整个屏幕从上到下所有像素点的个数。屏幕的左上角坐标为(0,0),右下角坐标为(639,479)。水平方向从左到右为 x 轴正向,垂直方向从上到下为 y 轴正向。Turbo C 的图形函数都是相对于图形屏幕坐标,即像素点来说的。

关于像素点的另外一个函数是:

```
int far getpixel(int x, int y);
```

调用该函数将获得当前像素点(x,y)的颜色值。

(2) 坐标位置函数

画点是与坐标相关的,有关坐标位置的函数如表 9.4 所示:

表 9.4 坐标位置函数

函数原型	函数功能
int far getmaxx(void);	返回 x 轴的最大值
int far getmaxy(void);	返回 y 轴的最大值
int far getx(void);	返回光标在 x 轴的位置
int far gety(void);	返回光标在 y 轴的位置
void far moveto(int x, int y);	移动光标到(x,y)点位置
void far moverel(int dx, int dy);	将光标从当前位置(x,y)移动到(x+dx,y+dy)的位置

2. 画线

(1) 画线函数

Turbo C 提供了一系列画线函数,简介如下:

- void far line(int x0, int y0, int x1, int y1);

画一条从点(x0, y0)到(x1, y1)的直线。

- void far lineto(int x, int y);

画一条从当前光标位置到点(x, y)的直线。

- void far linerel(int dx, int dy);

画一条从当前光标位置(x, y)到按相对增量确定的点(x+dx, y+dy)的直线。

- void far circle(int x, int y, int radius);

以(x, y)为圆心, radius 为半径画一个圆。

- void far arc(int x, int y, int stangle, int endangle, int radius);

以(x, y)为圆心, radius 为半径, 从 stangle 开始到 endangle 结束(用度表示)画一段圆弧线。在 Turbo C 中规定 x 轴正向为 0 度, 逆时针方向为正方向, 旋转一周为 360 度(其他有关函数也按此规定, 不再重述)。

- void ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);

以(x, y)为中心, xradius、yradius 分别为 x 轴和 y 轴半径, 从角 stangle 开始到 endangle 结束画一段椭圆线。当 stangle=0, endangle=360 时, 将画出一个完整的椭圆。

- void far rectangle(int x1, int y1, int x2, int y2);

以(x1, y1)为左上角, (x2, y2)为右下角画一个矩形框。

- void far drawpoly(int numpoints, int far * polypoints);

画一个顶点数为 numpoints, 各顶点坐标由 polypoints 给出的多边形。polypoints 整型数组必须至少有 2 倍顶点数个元素。每一个顶点的坐标都定义为(x, y), 并且 x 在前 y 在后。应当注意的是, 当画一个封闭的多边形时, numpoints 的值取实际多边形的顶点数加 1, 并且数组 polypoints 中第一个和最后一个点的坐标相同。

[例 9.4] 用 drawpoly() 函数画一个箭头。

```
#include <stdlib.h>
#include <graphics.h>
int main()
{
    int gdriver, gmode;
    int arw[16]={200,102,300,102,300,107,330,100,300,93,
    300,98,200,98,200,102};
    gdriver=DETECT;
    registerbgidriver(EGAVGA_driver);
    initgraph(&gdriver, &gmode, "");
    setbkcolor(BLUE);
    cleardevice();
    setcolor(12);          /* 设置作图颜色 */
    drawpoly(8, arw);      /* 画一箭头 */
    getch();
    closegraph();
}
```

(2) 设定线型函数

在没有对线的特性进行设定之前, Turbo C 使用其默认值, 即一个像素点宽的实线, 但 Turbo C 也提供了可以改变线型的函数。线型包括宽度和形状, 其中宽度只有两种选择(见表 9.5): 一个像素点宽和三个像素点宽; 而线的形状则有五种, 见表 9.6。

表 9.5 线的宽度

符号常数	数值	含义
NORM _ WIDTH	1	一个像素点宽
THIC _ WIDTH	3	三个像素点宽

表 9.6 线的形状

符号常数	数值	含义
SOLID _ LINE	0	实线
DOTTED _ LINE	1	点线
CENTER _ LINE	2	中心线
DASHED _ LINE	3	点划线
USERBIT _ LINE	4	用户定义线

下面是有关线型的设置函数:

• void far setlinestyle(int linestyle, unsigned upattern, int thickness);

该函数用来设置线的有关信息, 其中 thickness 指定线宽度, linestyle 指定线形状。

对于 upattern, 只有在 linestyle 选择 USERBIT _ LINE 时才有意义(选择其他线型时将参数 upattern 置为 0 即可)。这里 upattern 的 16 位二进制数码对应一个 16 个像素点的线段, 其每一位代表一个像素点, 如果其一位为 1, 则画线时在该像素点位置画点, 否则该像素点位置不画点。如 upattern 取值为 0xFFFF 时画线的形状将是实线, 而取值为 0x5555 时为点线。

• void far getlinesettings(struct linesettingstype far * lineinfo);

该函数将有关线的信息存放到由 lineinfo 指向的结构中, linesettingstype 结构定义如下:

```
struct linesettingstype {
    int linestyle;
    unsigned upattern;
    int thickness;
};
```

• void far setwritemode(int mode);

该函数规定画线的方式。如果 mode=0, 画线时将会把所画位置原来的画线信息覆盖(这是 Turbo C 的默认方式)。如果 mode=1, 则画线时用现在特性的线与所画之处原有的线进行异或(XOR)操作, 实际上画出的线是两者异或后的结果。因此当线的特性不变, 进行两次画线操作相当于没有画线。

[例 9.5] 线型设定和画线函数的例子。

```
#include <stdlib.h>
#include <graphics.h>
int main()
{
    int gdriver, gmode, i;
    gdriver=DETECT;
    registerbgidriver(EGAVGA _ driver);
    initgraph(&gdriver, &gmode, "");
```

```

    setbkcolor(BLUE);
    cleardevice();
    setcolor(GREEN);
    circle(320, 240, 98);
    setlinestyle(0, 0, 3);           /* 设置三点宽实线 */
    setcolor(2);
    rectangle(220, 140, 420, 340);
    setcolor(WHITE);
    setlinestyle(4, 0xaaaa, 1);     /* 设置一点宽用户定义线 */
    line(220, 240, 420, 240);
    line(320, 140, 320, 340);
    getch();
    closegraph();
}

```

9.1.5 填充

所谓填充就是用规定的颜色和图模(图形式样)填满一个封闭图形。

1. 轮廓填充

Turbo C 提供了一些先画出基本图形轮廓,再按规定图模和颜色填充整个封闭图形的函数。在没有改变填充方式时,Turbo C 将以默认方式填充。下面介绍这些函数。

- void far bar(int x1, int y1, int x2, int y2);

画出一个以(x1,y1)为左上角,(x2,y2)为右下角的矩形框,再按规定图模和颜色填充。

- void far bar3d(int x1, int y1, int x2, int y2, int depth, int topflag);

画出一个以(x1,y1)为左上角,(x2,y2)为右下角的矩形框作为底,depth 为深度的三维长方体,再按规定图模和颜色填充矩形框。长方体第三维(长度为 depth 的那一维)的方向不随任何参数而变,即始终为 45 度的方向。当 topflag 为非 0 时,画出一个三维的长方体;当 topflag 为 0 时,三维图形不封顶,实际上很少这样使用。

- void far pieslice(int x, int y, int stangle, int endangle, int radius);

画出一个以(x,y)为圆心,radius 为半径,stangle 为起始角度,endangle 为终止角度的扇形,再按规定方式填充。

- void far sector(int x, int y, int stanle, intendangle, int xradius, int yradius);

画出一个以(x,y)为圆心,分别以 xradius 和 yradius 为 x 轴和 y 轴半径,stangle 为起始角,endangle 为终止角的椭圆扇形,再按规定方式填充。

2. 设定填充方式

Turbo C 有四个与填充方式有关的函数。

- void far setfillstyle(int pattern, int color);

color 的值是当前屏幕图形模式时颜色的有效值。pattern 为填充式样,由表 9.7 指定:

表 9.7 填充式样 pattern 的规定

符号常数	数值	填充式样	符号常数	数值	填充式样
EMPTY_FILL	0	空心(背景色)	HATCH_FILL	7	直方网格
SOLID_FILL	1	实心(color 色)	XHATCH_FILL	8	斜网格
LINE_FILL	2	直线	INTTERLEAVE_FILL	9	间隔点
LTSLASH_FILL	3	斜线	WIDE_DOT_FILL	10	稀疏点
SLASH_FILL	4	粗斜线	CLOSE_DOT_FILL	11	密集点
BKSLASH_FILL	5	粗反斜线	USER_FILL	12	用户定义
LTBKSLASH_FILL	6	反斜线			

除 USER_FILL(用户定义填充式样)以外,其他填充式样均可由 setfillstyle()函数设置。选用 USER_FILL 作为函数 setfillstyle()的参数时,该函数对填充图模和颜色并不作任何改变。该符号常数可用于使用 getfillsettings()函数获取填充信息时。

• void far setfillpattern(char * upattern, int color);

设置用户定义的填充图模和颜色以供对封闭图形填充。

其中 upattern 是一个指向 8 个字节的指针。这 8 个字节定义了 8×8 点阵的图形。每个字节的 8 位二进制数表示水平 8 点,8 个字节表示 8 行,然后以此为模型向整个封闭区域填充。

• void far getfillpattern(char * upattern);

该函数将用户定义的填充图模存入 upattern 指针指向的内存区域。

• void far getfillsettings(struct fillsettingstype far * fillinfo);

获得当前使用的图模及颜色并存入结构指针变量 fillinfo 中。其中 fillsettingstype 结构定义如下:

```
struct fillsettingstype
{
    int pattern;    /* 现行填充模式 */
    int color;     /* 现行填充颜色 */
};
```

[例 9.6] 有关图形填充样式选择的例子。

```
#include <graphics.h>
main()
{
    char str[8]={10,20,30,40,50,60,70,80};    /* 用户定义图模 */
    int gdriver,gmode,i;
    struct fillsettingstype save;    /* 定义一个用来存储填充信息的结构变量 */
    gdriver=DETECT;
    initgraph(&gdriver,&gmode,"c:\\tc");
    setbkcolor(BLUE);
    cleardevice();
    for(i=0;i<13;i++)
```

```

    {
        setcolor(i+3);
        setfillstyle(i,2+i);           /* 设置填充类型 */
        bar(100,150,200,50);           /* 画矩形并填充 */
        bar3d(300,100,500,200,70,1);   /* 画长方体并填充 */
        pieslice(200, 300, 90, 180, 90); /* 画扇形并填充 */
        sector(500,300,180,270,200,100); /* 画椭圆扇形并填充 */
        delay(1000);                    /* 延时 1 秒 */
    }
    cleardevice();
    setcolor(14);
    setfillpattern(str, RED);
    bar(100,150,200,50);
    bar3d(300,100,500,200,70,0);
    pieslice(200,300,0,360,90);
    sector(500,300,0,360,100,50);
    getch();
    getfillsettings(&save);             /* 获得用户定义的填充模式信息 */
    closegraph();
    clrscr();
    printf("填充图模是:%d, 填充颜色是:%d", save.pattern, save.color);
    getch();
}

```

以上程序运行结束后,在屏幕上显示出当前填充图模和颜色的常数值。

3. 任意封闭图形的填充

以上是对一些特定的规则形状的封闭图形进行填充。对于任意封闭图形,Turbo C 也提供了一个填充函数,其调用格式如下:

```
void far floodfill(int x, int y, int border);
```

其中 x, y 为封闭图形内的任意一点。 $border$ 为边界的颜色,也就是封闭图形轮廓的颜色。点 (x, y) 和指定颜色的边界共同确定一个填充区域,调用了该函数后,规定颜色的图模将填满填充区域。

[例 9.7] 填充 `bar3d()` 所画三维长方体中其他两个未填充的面。

```

#include <stdlib.h>
#include <graphics.h>
main()
{
    int gdriver, gmode;
    struct fillsettingstype save;
    gdriver=DETECT;
    initgraph(&gdriver, &gmode, "");

```



```

    setbkcolor(BLUE);
    cleardevice();
    setcolor(LIGHTRED);
    setlinestyle(0,0,3);
    setfillstyle(1,14);           /* 设置填充方式 */
    bar3d(100,200,400,350,200,1); /* 画长方体并填充 */
    floodfill(450,300,LIGHTRED);  /* 填充长方体另外两个面 */
    floodfill(250,150,LIGHTRED);
    rectanle(450,400,500,450);    /* 画一矩形 */
    floodfill(470,420, LIGHTRED); /* 填充矩形 */
    getch();
    closegraph();
}

```

9.1.6 图形窗口和图形屏幕操作函数

1. 图形窗口操作

在图形方式下也可以在屏幕上某一区域设定窗口,其后有关图形的操作都将以这个窗口的左上角(0,0)作为坐标原点来进行,而且可以通过设置使窗口之外的区域为不可接触。这样所有的图形操作就被限定在这窗口内进行。

- void far setviewport(int xl, int yl, int x2, int y2, int clipflag);

设定一个以(xl,yl)像素点为左上角,(x2,y2)像素点为右下角的图形窗口,其中(xl,y1)和(x2,y2)是相对于整个屏幕的坐标。若 clipflag 为非 0,则设定的窗口以外部分不可接触;若 clipflag 为 0,则图形窗口以外可以接触,此时图形可以画到窗口以外的屏幕上。

- void far clearviewport(void);

清除现行图形窗口内的内容。

- void far getviewsettings(struct viewporttype far * viewport);

获得关于现行窗口的信息,并将其存于 viewporttype 类型的结构变量 viewport 中,其中 viewporttype 的结构定义如下:

```

struct viewporttype {
    int left, top, right, bottom;
    int cliplag;
};

```

2. 屏幕操作

除了前面介绍的清屏函数 cleardevice()以外,关于屏幕操作还有以下函数:

- void far setactivepage(int pagenum);

- void far setvisualpage(int pagenum);

这两个函数只适用于 EGA、VGA 以及 HERCULES 图形适配器。setactivepage() 函数是为图形输出选择激活页。所谓激活页是指后续图形操作的输出被写到 pagenum 指定的页面,该页面并不一定可见。setvisualpage()函数使 pagenum 所指定的页面变成可见页,页面从 0 开

始 (Turbo C 默认页)。如果先用 `setactivepage()` 函数在不同页面上画出若干帧图像,再用 `setvisualpage()` 函数依次交替显示这些页面内容,就可以实现一些动画的效果。

- `void far getimage(int xl, int yl, int x2, int y2, void far * mapbuf);`
- `void far putimage(int x, int y, void * mapbuf, int op);`
- `unsigned far imagesize(int xl, int yl, int x2, int y2);`

这三个函数用于将屏幕上的图像复制到内存,然后再将内存中的图像送回到屏幕上。首先通过函数 `imagesize()` 测试要保存左上角为 (xl, yl) , 右上角为 $(x2, y2)$ 的图形屏幕区域内的全部内容需多少个字节,然后再给指针 `mapbuf` 分配一个所测数大小的内存空间。通过调用 `getimage()` 函数就可将该区域内的图像保存在内存中,需要时可用 `putimage()` 函数将该图像输出到左上角为点 (x, y) 的位置上,其中 `putimage()` 函数中的参数 `op` 规定如何绘制内存中图像, `op` 取值于表 9.8 所示。

表 9.8 参数 `op` 的取值表

符号常数	数值	含义	符号常数	数值	含义
<code>COPY_PUT</code>	0	复制(覆盖屏幕图像)	<code>AND_PUT</code>	3	与屏幕图像与后复制
<code>XOR_PUT</code>	1	与屏幕图像异或复制	<code>NOT_PUT</code>	4	复制反像的图形
<code>OR_PUT</code>	2	与屏幕图像或后复制			

`imagesize()` 函数只能返回字节数小于 64K 字节的图像区域,否则将会出错,出错时返回 -1。

这些函数在图像动画处理、菜单设计技巧中非常有用。

[例 9.8] 下面程序模拟两个小球动态碰撞过程。

```
#include <stdio.h>
#include <graphics.h>
main()
{
    int i, gdriver, gmode, size;
    void * buf;
    gdriver=DETECT;
    initgraph(&gdriver, &gmode, "");
    setbkcolor(BLUE);
    cleardevice();
    setcolor(LIGHTRED);
    setlinestyle(0, 0, 1);
    setfillstyle(1, 10);
    circle(100, 200, 30);
    floodfill(100, 200, 12);
    size=imagesize(69, 169, 131, 231);
```

```

    buf=malloc(size);
    getimage(69, 169, 131, 231, buf);
    putimage(500, 269, buf, COPY_PUT);
    for(i=0; i<185; i++)
    {
        putimage(70+i, 170, buf, COPY_PUT);
        putimage(500-i, 170, buf, COPY_PUT);
    }
    for(i=0; i<185; i++)
    {
        putimage(255-i, 170, buf, COPY_PUT);
        putimage(315+i, 170, buf, COPY_PUT);
    }
    getch();
    closegraph();
}

```

9.1.7 图形模式下的文本输出

在图形模式下,可以使用标准输出函数,如 printf()、puts()、putchar()等函数输出文本到屏幕。输出结果均以白色为前景色,并且按 80 列 25 行的文本方式输出。

Turbo C 2.0 提供了一些专门用于图形模式的文本输出函数,这些函数可以用于文本的定位输出。

1. 文本输出函数

- void far outtext(char far *textstring);

该函数输出字符串指针 textstring 所指的文本到当前光标位置。

- void far outtextxy(int x, int y, char far *textstring);

该函数输出字符串指针 textstring 所指的文本到指定的(x,y)坐标位置。

2. 文本字体、字型和输出方式的设置

对于图形方式下的文本输出函数,可以通过 setcolor()函数设置输出文本的颜色。另外,还可以改变文本字体大小以及选择是水平方向输出还是垂直方向输出等输出方式。

- void far settextjustify(int horiz, int vert);

该函数调整文本字符串的输出位置。

对于使用 outtextxy(int x, int y, char far *str textstring) 函数所输出的字符串 textstring,其中哪个点对应于定位坐标(x, y)? 这在 Turbo C 2.0 中是有规定的。如果把一个字符串看成一个长方形的图形,在水平方向显示时,字符串长方形按垂直方向可分为顶、中、底三个位置,水平方向可分为左、中、右三个位置,两者结合就有 9 个位置,如图 9.1 所示。

settextjustify()函数的第一个参数 horiz 指出水平方向三个位置中的一个,第二个参数 vert 则指出垂直方向三个位置中的一个,两者就确定了其中一个位置。当规定了这个位置后,用 outtextxy()函数输出字符串时,字符串长方形的这个规定位置就对准函数中的(x, y)位置。

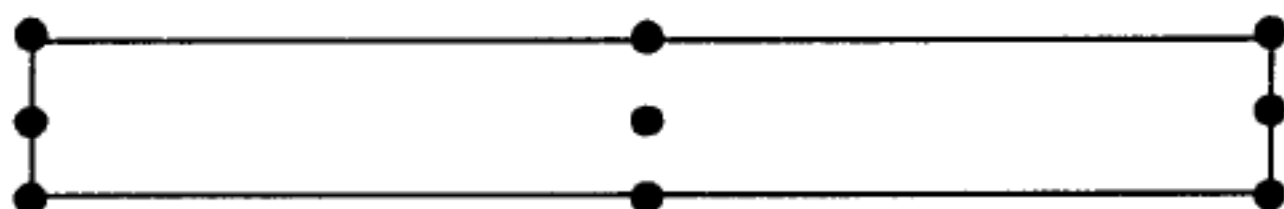


图 9.1 定位坐标(x,y)的九个字符串位置

而对用 `outtext()` 函数输出字符串时,这个规定的位置就置于当前光标所在的位置。参数 `horiz` 及参数 `vert` 的取值分别见表 9.9 和表 9.10。

表 9.9 参数 `horiz` 的取值表

符号常数	数值	含义
<code>LEFT_TEXT</code>	0	左位置
<code>CENTER_TEXT</code>	1	中位置
<code>RIGHT_TEXT</code>	2	右位置

表 9.10 参数 `vert` 的取值表

符号常数	数值	含义
<code>BOTTOM_TEXT</code>	0	底位置
<code>CENTER_TEXT</code>	1	中位置
<code>TOP_TEXT</code>	2	顶位置

• `void far settextstyle(int font, int direction, int charsize);`

该函数用来设置输出字符的字形(由 `font` 确定)、输出方向(由 `direction` 确定)和字符大小(由 `charsize` 确定)等特性。Turbo C 2.0 对该函数中各个参数的规定见表 9.11~表 9.13。

表 9.11 参数 `font` 的取值

符号常数	数值	含义
<code>DEFAULT_FONT</code>	0	8×8 点阵字
<code>TRIPLEX_FONT</code>	1	三重笔画字体
<code>SMALL_FONT</code>	2	小号笔画字体
<code>SANSERIF_FONT</code>	3	无衬笔画字体
<code>GOTHIC_FONT</code>	4	黑体笔画字体

表 9.12 参数 `direction` 的取值

符号常数	数值	含义
<code>HORIZ_DIR</code>	0	从左到右
<code>VERT_DIR</code>	1	从底到顶

表 9.13 参数 `charsize` 的取值

数值	字符大小	数值	字符大小
0(<code>USER_CHAR_SIZE</code>)	用户定义大小	6	84×48 点阵
1	8×8 点阵	7	56×56 点阵
2	16×16 点阵	8	64×64 点阵
3	24×24 点阵	9	72×72 点阵
4	32×32 点阵	10	80×80 点阵
5	40×40 点阵		

[例 9.9] 设置图形屏幕下文本输出和字体字型的例子。

```
#include <graphics.h>
#include <stdio.h>
int main()
{
    int i, gdriver, gmode;
```

```

char s[30];
gdriver=DETECT;
initgraph(&gdriver, &gmode, "");
setbkcolor(BLUE);
cleardevice();
setviewport(100, 100, 540, 380, 1);    /* 定义一个图形窗口 */
setfillstyle(1, 2);                    /* 绿色实填充 */
setcolor(YELLOW);
rectangle(0, 0, 439, 279);
floodfill(50, 50, 14);
setcolor(12);
settextstyle(1, 0, 8);                  /* 三重笔画字体, 水平放大 8 倍 */
outtextxy(20, 20, "Good Better");
setcolor(15);
settextstyle(3, 0, 5);                  /* 无衬笔画字体, 水平放大 5 倍 */
outtextxy(120, 120, "Good Better");
setcolor(14);
settextstyle(2, 0, 8);
i=620;
sprintf(s, "Your score is %d", i);      /* 将数字转化为字符串 */
outtextxy(30, 200, s);                 /* 指定位置输出字符串 */
setcolor(1);
settextstyle(4, 0, 3);
outtextxy(70, 240, s);
getch();
closegraph();
}

```

3. 用户对文本字符大小的设置

前面介绍的 `settextstyle()` 函数, 可以设定图形方式下输出文本字符的字形和大小。但对于笔画型字体(8×8 点阵字以外的字体), 只能在水平和垂直方向以相同的放大倍数放大。为此 Turbo C 2.0 专门提供了另外一个 `setusercharsize()` 函数, 它可以对笔画字体分别设置水平和垂直方向的放大倍数。该函数的调用格式为:

```
void far setusercharsize(int mulx, int divx, int muly, int divy);
```

该函数用来设置笔画型字的放大系数, 它只有在 `settextstyle()` 函数中的 `charsize` 为 0 (或 `USER_CHAR_SIZE`) 时才起作用, 并且字体应为函数 `settextstyle()` 中所规定的字体。当调用函数 `setusercharsize()` 之后, 每个显示在屏幕上的字符都以其缺省大小(8×8 点阵)乘以 `mulx/divx` 为输出字符宽, 乘以 `muly/divy` 为输出字符高。

[例 9.10] 函数 `setusercharsize()` 调用的例子。

```

#include <stdio.h>
#include <graphics.h>

```



```

int main()
{
    int gdriver, gmode;
    gdriver=DETECT;
    initgraph(&gdriver, &gmode, "");
    setbkcolor(BLUE);
    cleardevice();
    setfillstyle(1, 2);           /* 设置填充方式 */
    setcolor(WHITE);             /* 设置白色作图 */
    rectangle(100, 100, 330, 380);
    floodfill(50, 50, 14);       /* 填充方框以外的区域 */
    setcolor(12);                /* 作图色为淡红 */
    settextstyle(1, 0, 8);        /* 三重笔画字体, 放大 8 倍 */
    outtextxy(120, 120, "Very Good");
    setusercharsize(2, 1, 4, 1); /* 水平放大 2 倍, 垂直放大 4 倍 */
    setcolor(15);
    settextstyle(3, 0, 5);        /* 无衬笔画字体, 放大 5 倍 */
    outtextxy(220, 220, "Very Good");
    setusercharsize(4, 1, 1, 1);
    settextstyle(3, 0, 0);
    outtextxy(180, 320, "Good");
    getch();
    closegraph();
}

```

4. 登记连接时使用的字体

在使用非缺省字体(8×8 点阵字)时程序需要字体文件(*.CHR)的支持,对于四种笔画字体:三重笔画字体、小号字体、无衬线笔画字体和黑体,Turbo C 均提供了相应的字体文件:TRIP.CHR、LITT.CHR、SANS.CHR 和 GOTH.CHR。

为了使用方便,应该建立一个不需要字体文件就能独立运行的可执行图形程序,类似于图形驱动程序的登记,Turbo C 中规定用下述步骤(这里以黑体字为例)来建立:

(1)在 C:\TC 子目录下输入命令:BGIOBJ GOTH

此命令将字体文件 GOTH.CHR 转换成 GOTH.OBJ 的目标文件。此时屏幕将提示相应的字体函数 gothic _font 以供使用。

(2)在 C:\TC 子目录下输入命令:TLIB LIB\GRAPHICS.LIB+GOTH

此命令的意思是将 GOTH.OBJ 的目标模块装到 GRAPHICS.LIB 库文件中。

(3)在程序中黑体字的使用之前加上一句:

```
registerbgifont(gothic _driver);
```

该函数告诉连接程序在连接时把黑体字装入到用户的执行程序中。其函数原型如下:

```
int registerbgifont(void (*font)(void));
```

利用以上介绍的函数,可以在程序中方便地进行图形程序设计,如图形用户界面及计算结

果的图形输出。剩下的问题可能就是程序设计的技术了,这需要深入地学习和大量地积累编程经验。

9.2 中断处理

利用 C 语言进行程序设计可以实现许多通常只有汇编语言这种低级语言才能实现的功能,如中断处理,这也是 C 语言的一大特色。C 系统不仅支持行间汇编,用来实现中断处理,而且还提供了一系列的相关系统库函数来实现它,方便编程者使用。

中断是一种特殊类型的指令,它导致现程序的中止,在堆栈中保存系统现行状态,然后跳转到由中断号决定的中断处理过程。一旦中断处理过程结束,将恢复原来的被保存的系统状态,继续执行被中止的程序。中断是现代计算机发展中一个重要的技术,它能确保中央处理器在运行过程中随时地对外界发生的请求予以响应,在完成实时性的处理后又立即回到调用点继续被中断的工作。中断有两种基本类型:硬件产生的中断和软件产生的中断。

硬件中断是管理外部设备最灵活、效率最高的一种方式,它能使系统充分发挥中央处理器(CPU)的利用率,是通过中断控制器芯片来实现的。软件中断可以使我们的程序直接访问操作系统和系统资源,这极大地提高了应用程序的效能。这里我们仅关心软件中断。

8086 系列 CPU 允许通过 INT 指令(中断指令,或称为软件自陷指令)执行一个软件中断,当 CPU 在接受 INT 指令时,立即根据跟在指令后的数字(中断号)决定执行哪一个中断。例如,INT 21h 指令引起中断,21h 号中断服务程序(ISR)被执行。中断号用于查找中断服务程序的入口地址,该入口地址以段地址:位移量(称为中断向量,占 4 个字节)的形式保存在内存地址 0~3FFh 的 1 KB 大小的区域中,因此 8086 系列 CPU 最多支持 256 个中断号(中断向量)。这个存储区以中断号顺序依次存放中断向量表,例如,0 号中断向量存放在地址 0000:0000 中,而 1 号中断向量存放在地址 0000:0004 中,以此类推。在 PC DOS 系统中,给出的典型的 XT、AT 型 PC 机的中断分配表见表 9.14。

可以看出,中断向量主要分为三类:

BIOS 中断向量、DOS 中断向量和用户中断向量,它们分别用于 BIOS 和 DOS 功能调用,以及用户设计的中断服务程序。对于前两种,可以通过 C 语言提供的一组中断处理函数来实现;对于用户设计的中断例程,则需要用户设计中断服务程序来实现。许多实用程序如通讯程序和常驻内存程序都常常利用中断处理来实现。

9.14 中断分配

中断号	解释
0h ~ 1Fh	BIOS 中断向量
20h ~ 3Fh	DOS 中断向量
40h ~ 5Fh	扩充 BIOS 中断向量
60h ~ 6Fh	用户中断向量
70h ~ 77h	I/O 中断向量
78h ~ 7Fh	保留
80h ~ F0h	BASIC 中断向量
F1h ~ FFh	保留

值得注意的是由于 PC 兼容机品种繁多,再者随着微型计算机系统的发展,对于 40h~FFh 的中断分配不尽统一,如对于 386 以上的微机,系统基本上已不再保留 BASIC 中断。40h~FFh 这一区的中断也可称为自由中断,供系统或应用程序设置开发的中断例程使用,程序设计时若需使用,可参考相应手册。

9.2.1 中断的允许和禁止

中断可分为不可屏蔽中断和可屏蔽中断。需要立即响应的中断,如系统掉电是不可屏蔽的中断(NMI),由外设和软件请求产生的中断为可屏蔽中断(INTR)。对于可屏蔽中断只有当中断请求标志 $IF=1$ (表示允许中断)时,CPU 才响应中断,当 $IF=0$ (表示禁止中断)时,CPU 不响应中断请求,因此在程序中可以控制对 INTR 的响应,在 Turbo C 中使用:

```
void disable(void);      禁止中断
void enable(void);       允许中断
```

来设置中断请求标志 IF。这两个宏在系统头文件 dos.h 中定义。

同一时刻有多个中断请求时,系统总是响应优先级较高的中断,优先级由高到底的顺序通常是:CPU 内部中断(如除法错,溢出等)、不可屏蔽中断、可屏蔽中断、单步中断。

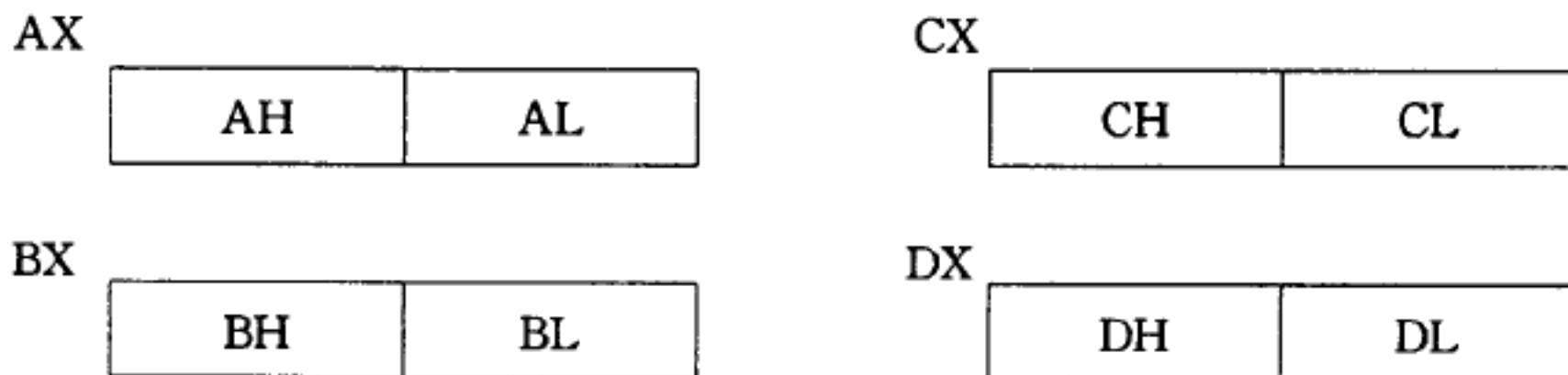
当某个中断发生时,它会设置中断请求标志 IF 为 0,禁止其他可屏蔽中断的请求,因此,在进入中断程序以后,应首先允许中断,防止某些重要的中断被禁止而引起系统运行不正常。

9.2.2 DOS 与 BIOS 功能调用

在 C 语言中,中断处理的过程为:

1. 通过寄存器约定,把参数赋给寄存器,以供服务程序使用;
2. 通过中断处理函数发出中断请求,并从中断返回;
3. 从约定的寄存器中取出返回值(如果需要的话)。

寄存器是 CPU 中的存储空间,通常分为通用寄存器 (AX、BX、CX、DX),指针寄存器 (SP、BP、IP),索引寄存器 (SI、DI),段寄存器 (CS、DS、SS、ES)和标志寄存器 (FLAGS)。其中通用寄存器又可分为高位和低位寄存器,结构如下:



每个中断访问一个特定的功能类,BIOS 功能调用提供低级过程,DOS 功能调用提供高级功能。AH 寄存器的值决定功能的内容,如果需要附加的信息就将它们放在 AL、BX、CX、DX 寄存器之中。

在头文件 dos.h 中定义了以下数据结构用于程序与寄存器之间的传值:

- 定义段寄存器

```
struct SREGS
{
    unsigned int es, cs, ss, ds;
}
```

- 定义 16 位寄存器

```

struct WORDREGS
{
    unsigned int ax, bx, cx, dx, si, di, cflag, flags;
}
• 定义 8 位寄存器
struct BYTEREGS
{
    unsigned char al, ah, bl, bh, cl, ch, dl, dh;
}
• 定义 16 位和 8 位寄存器的联合结构
union REGS
{
    struct WORDREGS x;
    struct BYTEREGS h;
}
• 定义寄存器包
struct REGPACK
{
    unsigned int r_ax, r_bx, r_cx, r_dx;           /* 通用寄存器 */
    unsigned int r_bp;                             /* 基址寄存器 */
    unsigned int r_si, r_di;                       /* 索引寄存器 */
    unsigned int r_ds, r_es;                       /* 段寄存器 */
    unsigned int flags;                            /* 标志寄存器 */
}

```

利用以上数据结构,就可以在使用 C 语言进行程序设计时调用 C 系统提供的中断处理函数来实现中断调用。下面是几个中断处理函数,它们用于 8086 系列 CPU 的中断调用,函数原型在头文件 dos.h 之中。

• int int86(int intno, union REGS * in, union REGS * out);

用参数 in 加载 CPU 中的寄存器,发出 intno(中断号)中断,并储存结果到 out 中。

该函数用于 T(tiny:微型)、M(medium:中型)、S(small:小型)编译模式。

• int int86x(int intno, union REGS * in, union REGS * out, struct SREGS * seg);

功能同 int86(),但具有参数 seg(段地址)。

该函数用于 C(compact:紧凑型)、L(large:大型)、H(huge:巨型)编译模式。

• int intr(int intno, struct REGPACK * preg);

功能同 int86(),但用参数 preg(寄存器包)加载和接收寄存器数据。

该函数仅用于 Turbo C。

INT 21h 的 DOS 功能调用是一个极其重要的软件中断,DOS 的内核即是由它组成的。它可分为设备管理、文件管理、目录操作及其他功能等几个部分。该中断号下有许多子功能,调用时使用功能号来加以区别,在进行 INT 21h 中断调用时,功能号将被送入寄存器 AH 中。表 9.15 例举部分较简单的 21h 号中断功能,更多内容请查阅 DOS 手册。

表 9.15 部分 DOS 系统功能调用(中断号=21h,AH=功能号)

功能号	功能	入口参数	出口参数
01h	字符键盘输入(回显)		AL=输入字符
02h	在屏幕上显示一个字符	DL=输出字符	
0Bh	检查键盘状态		AL=0 表示无键入;否则 AL=FFh
09h	打印字符串	DS:DX=字符串首地址 字符串以 \$ 结尾	
41h	删除文件	DS:DX=字符串首地址	
47h	取当前目录路径名	CL=盘号,DS:SI=串地址	
1Ah	设置磁盘传送区	DS:DX=磁盘传送区首地址	
4Eh	查找匹配的第一个文件	DS:DX=字符串首地址 CX=属性	出错时 AX=错误码
4Fh	查找下一个文件		出错时 AX=错误码
25h	设置中断向量	AL=中断号 DS:DX=中断向量地址	
2Ch	取系统时间		CX:DX=时间
2Dh	置系统时间	CX:DX=时间	AL=0 表示操作成功;否则 AL=FFh

应用于 DOS 的 INT 21h 中断调用,C 系统提供了以下函数来实现:

- int bdos(int dosfun, unsigned dx, unsigned al);

DOS 功能调用。其中参数 dosfun 是功能号,参数 dx,al 值将赋给相应寄存器,返回 AX 之值。用于 T、S、M 编译模式。

- int intdos(union REGS *in, union REGS *out);

功能同上。用参数 in 中的值加载 CPU 寄存器,结果由参数 out 传出。

- int intdosx(union REGS *in, union REGS *out, struct SREGS *seg);

同 intdos(),但具有段地址 seg,因而可用于 C、L、H 编译模式。

- int bdosptr(int dosfun, void *dsdx, unsigned al);

使用指针 dsdx(段:偏移量)对寄存器 DS:DX 赋值,其余同 bdos()。

[例 9.11] 用中断方式输出字符串。

```
#include "dos.h"
main()
{
    char *s="Hello, World! $"; /* 输出字符串,需以 '$' 结尾 */
    union REGS reg;
    reg.h.ah=0x9;               /* 功能号为 09h,将被赋给寄存器 AH */
    reg.h.dx=(unsigned)s;        /* 字符串 s 的首地址将被赋给寄存器 DX */
    intdos(&reg,&reg);           /* DOS 功能调用 */
}
```

程序运行后屏幕上输出如下:

Hello, World!

此例可以用中断调用函数 `int86()` 来代替 `intdos()`:

```
int86(0x9, &reg, &reg);
```

还可以将主函数体改写成只有一条语句的形式:

```
bdos(0x09, (unsigned)"Hello, World! $", 0);
```

或者: `bdosptr(0x09, "Hello, World! $", 0);`

还有其他的表述形式。

[例 9.12] 屏幕列写工作目录下的文件(DIR 命令)。

```
#include "dos.h"
main()
{
    char pp[44];
    bdos(0x1a, (unsigned)pp, 0);          /* 设置磁盘传送缓冲区 */
    bdos(0x4e, (unsigned)"*. *", 0);      /* 查找第一个文件 */
    printf("%s\n", &pp[30]);
    for(;;)
    {
        if(18==bdos(0x4f, 0, 0))break; /* 查找下一个文件, AX 等于 18 时结束 */
        printf("%s\n", &pp[30]);
    }
}
```

[例 9.13] DOS 功能调用, 中断一个无限循环。

```
#include "dos.h"
#include "stdio.h"
main()
{
    for(;;)
    {
        putchar('a');
        if((char)bdos(0xb, 0, 0))break;
    }
}
```

以上程序执行后将不断地在屏幕上显示字符“a”, 直到用户敲击一次键盘, 此时将结束这个无限循环, 退出程序。这对于用户在程序运行中途干预程序流程提供了一个有效的手段。

DOS 功能调用(INT 21h)包含近百个系统子功能, 内容非常丰富。利用它可以在程序中实现系统级的功能, 这对于开发完善的应用程序, 将程序与操作系统融为一体是十分有帮助的。

BIOS 功能调用提供更为基本的 I/O 功能, 如键盘、视频、磁盘、打印机等的输入输出及控制操作, 下面仅介绍中断 16h(键盘 I/O)的 0 号功能调用(读键盘扫描码)的应用。

程序设计时对键盘输入的识别是必不可少的, 普通键进行键入时可以得到一个 8 位(1 字节)的字符 ASCII 码, 但对于功能键及其他特殊键, 它们的字符码都是 0, 不能进行区分。此时

就需要借助于 2 字节的键盘扫描码来进行识别,扫描码包括两个部分:低字节为字符码(对于普通字符即为 ASCII 码),高字节为位置码。特殊键的字符码虽然相同,但位置码却是不一样的。调用中断 16h 后,位置码在 AH 中,字符码在 AL 中。

[例 9.14] 下面程序用来识别键盘输入。

```
#include "stdio.h"
#include "dos.h"
int get _key()
{
    union REGS r;
    r.h.ah=0;                      /* 0 号功能调用 */
    return int86(0x16, &r, &r);    /* 16 号中断调用 */
}
main()
{
    union scan { int c; char ch[2]; }sc;
    do
    {
        sc.c=get _key();
        if(sc.ch[0]==0)
            printf("\n 特殊键:%d\n",sc.ch[1]);
        else
            putchar(sc.ch[0]);
    }while(sc.ch[0]! = 'q');
}
```

Turbo C 提供了几个函数用于 BIOS 功能调用,它们都在 bios.h 中定义:

- 检查系统配置(中断 11h)

```
int biosquip(void);
```

- 检测存储器容量(中断 12h)

```
int biosmemory(void);
```

- BIOS 磁盘服务函数(中断 13h)

```
int biosdisk(int ah, int dl, int dh, int ch, int cl, int al, void * buf);
```

- 串行端口服务函数(中断 14h)

```
int bioscom(int ah, char al, int dx);
```

- 键盘服务函数(中断 16h)

```
int bioskey(int ah);
```

- 打印服务函数(中断 17h)

```
int biosprint(int ah, int al, int dx);
```

- BIOS 时钟服务函数(中断 1Ah)

```
int biostime(int ah, long time);
```

以上函数实现相应的 BIOS 中断,若该中断具有子功能,其子功能号由参数 ah 来给出。

[例 9.14]中的函数 `get_key()`可改写如下:

```
int get_key()
{
    return bioskey(0);
}
```

9.2.3 中断服务程序

用户可以设计中断服务程序以实现自己的软件中断。它们一般作为设备驱动程序,或根据需要增补、改进、替换系统中断。设计中断服务程序可按如下步骤进行:

1. 把作为中断服务程序的函数说明为 `interrupt` 类型,并且无返回值,如:

```
void interrupt myintfun();
```

这里说明用户设计的函数 `myintfun()`为中断服务函数,赋予中断号后就可以通过该中断号调用它。

2. 利用中断向量安装中断服务函数

这里可以利用 Turbo C 提供的库函数 `setvect()`来实现,其函数原型在 `dos.h` 中,定义如下:

```
void setvect(int intno, void interrupt (*pt)());
```

该函数将 `pt` 所指向的函数安装为中断号为 `intno` 的中断服务函数,如函数调用:

```
setvect(0x61, myintfun);
```

将 `myintfun` 安装到中断向量表中,指定其中断号为 `61h`。

3. 通过中断号。可以用类似 DOS 或 BIOS 中断调用方法调用用户设计的中断服务程序,如:

```
int86(0x61,0,0);
```

除了前面介绍的 `int86()`系列的中断调用函数,Turbo C 还提供更为简洁的函数 `geninterrupt()`进行中断服务程序的调用,其函数原型在 `dos.h` 中,定义如下:

```
void geninterrupt(int intno);
```

参数 `intno` 指定中断号,它可以发出任何类型的中断请求。对于中断服务程序需要的参数,可以使用伪寄存器变量来直接传给对应的寄存器,伪寄存器变量由寄存器名(大写字母)之前加下划线构成,如 `_AX`、`_BX`、`_AL`、`_AH`、`_DI`、`_SS` 等等。程序设计时对它们进行赋值,实际上是将该值加载到相应的寄存器。

相应于 `setvect()`安装中断服务函数,下面函数获取中断向量:

```
void interrupt (*getvect(int intno))();
```

该函数返回与中断号 `intno` 对应的中断矢量(4 个字节),例如:

```
void interrupt (*pt)();
pt=getvect(0x61);
```

以上语句执行后 `pt` 便指向 `61h` 中断的入口地址。

用户中断服务程序设计是比较复杂的,主要是设计过程有比较严格的要求和较多的注意事项。作为简介,下面给出一个并不完善的例子。应该注意的是,如果程序占用了系统中断向量,则在程序退出前应该恢复这些中断向量,以保证系统的正常运行。

[例 9.15] 下面程序修改系统定时器中断 1Ch, 使屏幕右上角显示一个一位的计数器。

```
#include "dos.h"
#include "conio.h"
#define ATTR 0x7900          /* 设置写屏字符的背景前景色 */
void interrupt (*oldhandler)();
void interrupt myint()       /* 用户定义中断服务函数 */
{
    unsigned int (far *screen)[80];
    static int count;
    screen=(unsigned int (far *)[80])MK_FP(0xB800,0); /* 屏幕缓冲区 */
    count++;
    count%= 10;
    screen[0][79]=count+'0'+ATTR; /* 直接写屏 */
    oldhandler(); /* 原中断处理程序调用 */
}
main()
{
    oldhandler=getvect(0x1C); /* 获取原定时器中断向量 */
    setvect(0x1C,myint); /* 修改定时器中断为用户中断 */
    getch(); /* 等待用户按键以结束程序 */
    setvect(0x1C,oldhandler); /* 恢复原定时器中断 */
}
```

该程序运行过程中, 屏幕右上角将出现一个灰底蓝字循环变化的数字, 直到用户有击键动作, 结束程序。程序中还使用了一个宏 MK_FP(), 用于复合形式一个 32 位的 far 指针指向屏幕缓冲区, 这里 0xB800:0000 对应彩显视屏缓冲区, 若是单色显示器, 应将 0xB800 改为 0xB000。

宏 MK_FP() 的调用格式如下:

```
void far *MK_FP(unsigned segment, unsigned offset);
```

参数 segment 指定段地址, offset 指定偏移量, 返回一个 32 位的 far 指针。

对应地, 下面两个宏用于取段地址和偏移量:

取段地址: unsigned FP_SEG(void far *point);

取偏移量: unsigned FP_OFF(void far *point);

本章虽以实用程序设计为题, 但就其内容而言距实用有着很大的差距, 也只涉及了图形处理与中断处理两方面。希望这些内容对从 C 语言的初步学习过渡到应用程序设计, 能起到基础的引桥作用。

* 第 10 章

C++ 简介

Visual C++ 是美国微软公司开发的一种面向对象编程语言, 适宜于编制各式各样的软件。尤其适用于开发中、大型程序项目。使用 Visual C++ 的集成化开发环境, 能大大缩短开发时间, 减少开发费用, 使软件具有可靠性、重用性、扩充性与维护性。

10.1 C++ 的新特征

10.1.1 C++ 的输入/输出

1. 引例

首先, 我们从一个简单的 C 与 C++ 例子入手来看 C++ 与 C 语言的差别及 C++ 的编程风格。

[例 10.1] 输入你的名字, 并在屏幕上显示所输入的名字。

用 C 语言编程:

```
#define N 8
#include <stdio.h>
#include <stdlib.h>
void main()
{
    char * name;
    name=(char *)malloc(N * sizeof(char));
    printf("输入你的名字: ");
    scanf("%s", name);
    printf("你的名字是%s\n", name);
}
```

运行结果:

输入你的名字: 安丽
你的名字是安丽

用 C++ 语言编程:

```
const int N=8;
#include <iostream.h>

void main()
{
    char * name;
    name=new char[N];
    cout<<"输入你的名字: ";
    cin>>name;
    cout<<"你的名字是"<<name
    <<endl;
}
```

运行结果同左。

2. 说明

比较上述两个程序,可以看出,C++语言在输入输出以及对指针动态分配空间方面做了较大的改进。

(1)C++语言的输入/输出是通过流实现的,这将在第 6 节作详细介绍。读者目前只需知道 C++是自带输入/输出的,并能根据数据类型自动地使用合适的输入/输出方式。

(2)在第一行的头文件中,C 语言使用的是标准输入输出头文件 `stdio.h`,而 C++使用的则是输入输出流头文件 `iostream.h`,它体现了 C++的输入/输出风格。

(3)在常量定义方面,C 语言用“`#define N 8`”,而 C++则用“`const int N=8;`”,后者是语句,前者不是。

(4)在定义字符指针 `name` 后,C 语言要用内存动态分配函数 `malloc()`给指针分配 `N` 个 `char` 字节的内存地址,即“`name=(char *)malloc(N*sizeof(char));`”,而 C++语言则用 `new` 操作符进行动态内存分配,即“`name=new char[N];`”,显然,C++比 C 简单得多。

(5)在输出方面,C 语言使用 `printf()`函数,且要在参数表中指定输出格式,而 C++则只要使用“`cout<<`”就能把后面的内容送到标准输出设备(这里是显示器),并能自动地进行格式转换。

(6)在输入方面,C 语言中使用 `scanf()`函数进行输入,并要指定输入格式,而在 C++语言中,则只要使用 `cin>>`就能把输入设备(键盘)接收到的数据存入后面的变量。显然,C++的输入/输出比 C 简单、方便得多,但一定要加上包含头文件 `iostream.h`,而不是 `stdio.h`。

C 与 C++的比较见表 10.1。

表 10.1 C 与 C++的基本比较

语言 类别	C		C++
输入/输出	实现	流	流
	头文件	<code>stdio.h</code>	<code>iostream.h</code>
	输入	<code>scanf</code> 函数	<code>cin>></code>
	输出	<code>printf</code> 函数	<code>cout<<</code>
动态分配	<code>malloc(类型) / calloc(n,类型)</code>		<code>new 类型/new 类型(size)</code>
换行	<code>'\n'</code>		<code>endl</code>
字符常量	<code>#define</code> 字符常量名 常量		<code>const 类型 字符常量名=常量</code>
注释	形式	<code>/* 注释内容 */</code>	<code>// 注释内容</code>
	性质	多行	单行
程序扩展名	<code>.C</code>		<code>.CPP</code>
组成之一	函数		方法/函数

3. 应用举例

[例 10.2] 输入一个圆的半径,计算圆的面积。

C 语言程序

```
#include <stdio.h>
```

C++语言程序

```
#include <iostream.h>
```

```
#define PI 3.14159
void main()
{
    float Radius, Area;
    printf("输入圆的半径:");
    scanf("%f", Radius);
    Area=PI * Radius * Radius;
    printf("Area=\n", Area);
}
```

运行结果:

```
输入圆的半径:2
Area=12.5664
```

```
const float PI=3.14159f;
void main()
{
    float Radius, Area;
    cout<<"输入圆的半径:";
    cin>>Radius;
    Area=PI * Radius * Radius;
    cout<<"Area="<<Area<<endl;
}
```

10.1.2 C++的函数原型

在C++中,一定要使用函数原型,以使C++进行更强的类型检查,在编译阶段就发现函数调用表达式中可能存在的问题。如果缺少任何一个函数原型,C++都不能对其进行编译。

函数原型既标出函数的返回类型,也标出该函数的类型与个数。

C++编译器从一个函数定义中抽取该函数的原型。用户也可在程序中用说明语句来说明一个函数原型,函数说明语句的一段形式为:

```
类型 函数名(类型 1,类型 2,...,类型 n);
```

例如:

```
float fsum(float, float);
```

或:

```
类型 函数名(类型 1 参数 1,类型 2 参数 2,...,类型 n 参数 n);
```

例如:

```
float fsum(float a, float b)
```

在这里,名字a和b对编译器无意义,但选用恰当参数名,则可起到说明参数含义的作用,有助于用户了解函数的用法,这种风格与含义在C语言的函数说明中也是相同的。

10.1.3 C++函数的缺省参数

在C++中,函数调用时引进了一种缺省参数。缺省参数就是不要求设定该参数,而是由编译器在需要时给该参数赋予预先设定的值。当要传递一个与预先设置不同的值时,必须显式地指明,缺省参数是在函数原型中说明的,如:

```
int Name(char * n, char * n1="", char * n2="", char * n3="");
```

这种方式表明:①缺省的参数不管有几个,均需放在参数序列的最后;②在实际调用函数Name()时,调用函数可以忽略参数n1、n2、n3;③如果一个缺省的参数需要指明一个特定的

值,则在它之前的所有参数均需赋值,在上例中,如果给参数 n2 赋以 C# 值,则必须同时对 n1 赋值。如:

```

           n1      n2      n3
           ↑       ↑       ↑
stafus=Name("Good", "C++", "C#", "Java");

```

10.1.4 C++ 的 new 与 delete

1. 使用形式

使用指针时,可用动态分配内存。在 C 语言中,动态分配内存是用系统函数 malloc() 和 calloc() 来实现,并用 free() 函数来释放内存,而在 C++ 中,则是分别用 new 和 delete 运算符来实现动态分配内存和释放内存。

(1) 使用 new 运算符对指针分配内存的形式为:

指针=new 类型;

(2) 对数组指针分配内存的形式为:

数组指针=new 类型[数组长度];

使用 new 的最大优点是不需进行类型转换,如对含有 6 个整数的数组分配内存,可写成:

```

int *ipt;
ipt=new int[6];

```

或写成:

```

int *iptr=new int[6];

```

(3) 使用 delete 释放 new 分配内存的形式为:

delete 指针;

2. 应用示例

[例 10.3] new 和 delete 的用法。

```

#include <iostream.h>
void main()
{
    float *fPtr;
    fPtr=new float;
    *fPtr=9.18f;
    cout<<*fPtr;
    delete fPtr;
    cout<<endl<<endl;
}

```

运行结果:

9.18

上例说明:

(1) *fptr=9.18f; 是对变量赋初值,也可在用 new 分配内存的同时对其赋初值,如:

```

float *fptr=new float(9.18f);

```

(2)与C语言中使用 malloc()一样,当分配内存失败时,new 返回空指针 0,如:

```
float * fptr=new float[1000];
if(fptr==0)
    cout<<"分配内存失败!";
```

10.1.5 C++的内联函数

1. 内联函数的定义

在函数定义前面加上关键字 inline 即构成内联函数。形式为:

```
inline 返回类型 函数名(参数类型说明表)
{
    函数体
}
```

这样,C++编译器遇到对该函数进行调用时,就用该函数的函数体进行替换,即转向执行该函数。

内联函数的定义(包括说明与代码)必须出现在每一调用函数的源文件中,否则,编译器因无法访问其代码而不能实现内联。使用内联函数会使目标代码的生成速度和目标代码的大小增加。内联函数与宏及其展开相类似,但在概念上不是一回事,必须加以区分。

2. 应用示例

[例 10.4] 求两个数中最大的数。

```
#include <iostream.h>
inline double max(double a, double b)
{
    return (a>b)? a:b;
}
void main()
{
    double x, y, Max;
    cout<<"请输入两个数:";
    cin>>x>>y;
    Max=max(x, y);
    cout<<"Max="<<Max<<endl<<endl;
}
```

运行结果:

```
请输入两个数: 1.24 10.25
Max=10.25
```

10.1.6 C++的引用

1. 引用的含义与用法

引用是C++对C的一个重大改进,所谓引用就是给变量取一个别名,别名和原变量共用一个地址。因而,不管对别名还是对原变量进行修改,实际上都是对同一地址的内容进行修改,也就是说原变量和别名总具有同样的值。

C++使用引用运算符&来定义引用,如果这个引用不作为函数的参数或返回值,则在说明时必须初始化,其形式为:

类型 & 别名=初值;

或: 类型 变量名=初值;

类型 & 别名=变量名;

例如:

```
float data=4.21; // data 变量必须赋以初值
```

```
float &ref=data; /* &ref 是一个 float 型别名的引用,所以要用 float 型的变量 data
对其赋初值,称 float & 为别名 ref 的引用类型 */
```

2. 应用示例

[例 10.5] 引用的使用实例。

```
#include <iostream.h>
void main()
{
    int iData=918;
    int &iRef=iData;
    iRef+=100;
    cout<<"iData="<<iData<<endl;
    iData+=421;
    cout<<"iRef="<<iRef<<endl<<endl;
}
```

运行结果:

```
iData=1018
```

```
iRef=1439
```

(1)在上例中,“int iData=918;”定义 iData 为 int 类型变量;而“int &iRef=iData;”则定义 iRef 为 int 型变量 iData 的引用,这表示,不管是对 iData 或 iRef 操作,均是对同一存储单元(开始存放 918 值的单元)的内容进行操作。

(2)赋给引用的初值,必须也具有初值的变量,若是一个常量或一个使用 const 修饰的符号常量,那么,编译器首先建立一个临时变量,即引用变量,然后将该常量放入临时变量,因此,对引用的操作就是对这个临时变量操作。如:

```
const int iData=918;
int & iRef=iData;
cout<<iRef ;      //输出 918
```



```

iRef += 421;
cout << iRef;      //输出 1439
cout << iData;     //输出 918

```

(3)若所说明的引用类型与初始化时所用的变量类型不一致,则编译器也建立一个与引用类型相同类型的临时变量,然后将该变量的值转换成引用类型的值,并放入临时变量中,对引用的访问同样是对临时变量的访问。

(4)由于引用与原变量具有同一地址,所以引用不是新变量,因而不能用引用说明引用,也不能说明数据类型为引用数组或指向引用的指针,但可以说明指针的引用。如:

```

int * ipt1;
int * &Refptr=ipt1; //Refptr 是指针变量 ipt1 的引用
int i;
Refptr=&i; //ipt1 指向变量 i

```

(5)在C++中,引用主要用于建立参数的引用(地址)传递方式,通过引用传递方式,可以不使用指针就可改变实参之值。

[例 10.6] 编写函数,交换两个整型变量的值。

```

#include <iostream.h>
void swap(int &a, int &b)  //a 是 a1 的引用, b 是 b1 的引用
{
    ↑      ↑
    a^=b;  a1    b1
    b^=a;
    a^=b;
    cout<<"交换后: a="<<a<<"\tb="<<b<<endl<<endl;
}
void main()
{
    int a1=918, b1=421;
    cout<<"交换前: a="<<a1<<"\tb="<<b1<<endl;
    swap(a1, b1);
}

```

运行结果:

```

交换前: a=918      b=421
交换后: a=421      b=918

```

在上例中,当在主函数中调用 swap 时,实参 a1 和 b1 分别传递给各自的引用 a 与 b,故在函数 swap 中,对 a 和 b 的访问就是对 a1 和 b1 的访问,因而,函数 swap 改变了 main 函数中变量 a1 和 b1 之值。

10.1.7 C++面向对象编程基础

面向对象的程序设计在软件设计除了具有结构化程序设计的优点和机制外,还引入了若干能反映事物本质的新概念,如类、封装、继承与多态性,这将避免在结构化程序设计中,由于代

码与数据分离,而带来的程序可维护性差、难以修改以及不可重用等缺点,从而大大推动了软件开发的进程。在面向对象程序设计中,我们将面对的是对象,而不是面对过程,这是一个全新的程序设计概念。对象是系统中最基本的运行实体,对象中封装了描述该对象的特殊属性(数据)和行为方式(方法)。整个应用程序由各种不同类型的对象组成,各对象既是独立的实体,又可通过消息(即让对象以某种方式进行操作的请求)相互作用,对象中的方法决定要向哪个对象发消息,发什么消息,以及收到消息时如何进行处理。对象的内部结构如图 10.1 所示。注意:为与 C 语言比较,在 C++ 中,我们把“方法”、“函数”统为一体,用函数描述,这样我们从 C 语言角度来理解,易于掌握。但作为面向对象的含义,它又必须是处理数据的方法,因此,下面描述中,根据需要,时而用方法,时而用函数。

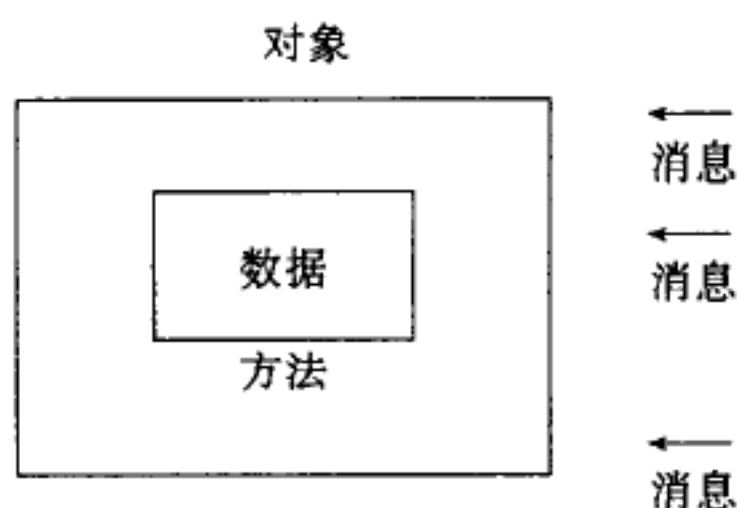


图 10.1 对象的内部结构

1. 封装

面向对象程序设计方法,是把数据及处理这些数据方法封装到一个类中。类其实是 C++ 的一种数据类型,而用类定义的变量称为对象(或称对象变量、实例、实例变量)。把数据及处理数据的方法封装在一起,构成一个对象的过程,称之为封装。在对象内只有属于该对象的函数成员才能存取该对象的数据成员,这样,其他函数就不能访问该对象的数据成员,从而可达到保护和隐藏数据的目的。

C++ 封装的基本单元称为类,是进行封装和数据隐藏的工具,类是逻辑上有关的方法及其数据的集合,它主要不是用于执行,而是提供所需要的资源。

2. 继承

在面向对象程序设计中,继承是指在已有类(称父类)基础上创建一个新类(称派生类)。用户通过增加、修改或替换父类中的方法成员产生派生类,以便对父类进行扩充,而派生类(子类)自动拥有父类的所有对象与方法,然后再根据需要给派生类添加所需的数据成员和方法,这就是类有分类概念。使用分类后,每一层的对象只需定义属于它本身的性质,其他性质就可从上一层继承。合理地使用继承可以减少很多的重复劳动。如果类实现了一个特别的功能,则它的派生类就可重复使用这个功能,而无需重新编程。

一个不由任何类派生的类称为基类,一个派生类的最近的上层类称为该类的父类(或超类),从某类派生出来的类称为该类的子类。类间的层次关系如图 10.2 所示。

一个类从派生它的基类到它自身可能要经过好几个层次。类不仅能继承其超类的所有对象和方法,而且还能继承从它的基类到它自身之间经过的所有层次上的类的对象和方法。

继承与封装具有很好的合作性。如果一个给定的类封装了某些属性,则它的任何子类将继承这些属性并可添加其特有的属性。

3. 多态性

多态性意味着一个对象具有多个面孔,允许其中方法接受不同类型参数的传递,实现不同

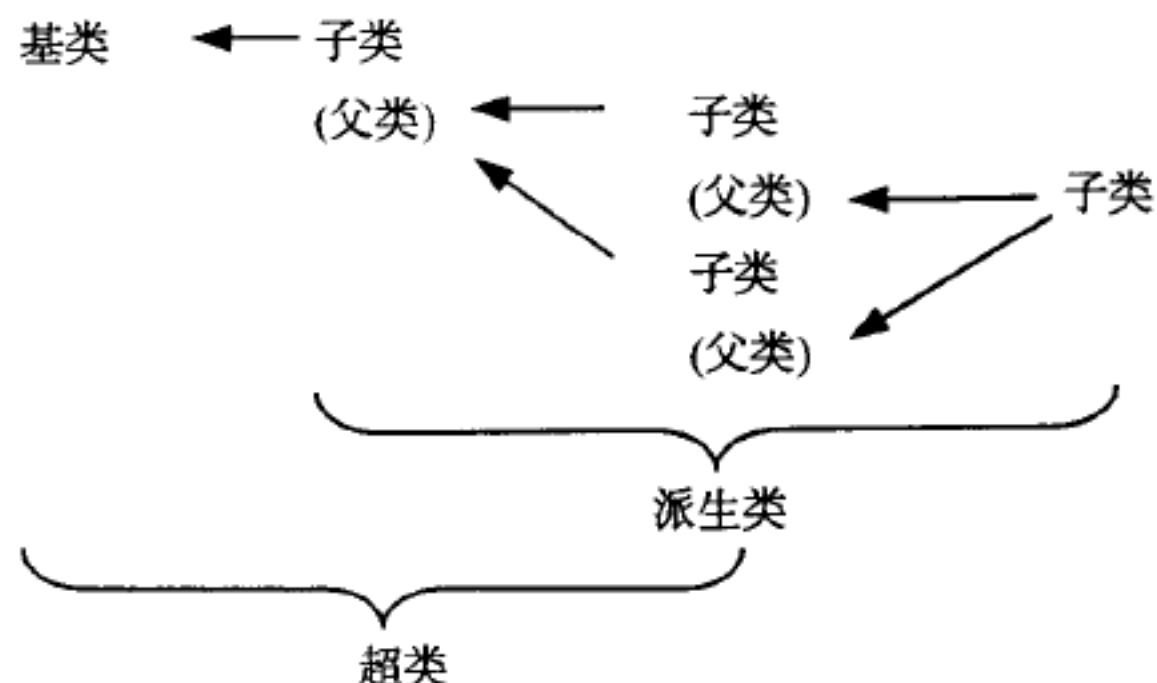


图 10.2 类间的层次关系

的功能。也就是说,调用相同函数而被不同的对象接受时,可以导致完全不同的行为,这种现象称为多态性。

例如,可能有两个方法都叫 `print()`,一个方法是在屏幕上显示字符串,另一个方法则是在屏幕上显示一个位图。当调用 `print()` 时,到底激活哪一个方法取决于调用的参数是字符对象还是位图对象。

多态性有时也指函数重载,即同一函数名可以对应多个函数的实现。函数重载允许一个程序内说明多个名称相同的函数,这些函数可以完成不同的功能,并可带有类型不同、数目不等的参数及返回值,编译器从调用函数的参数类型及个数上区别函数应实现的功能。函数重载是类以统一方式处理不同数据类型的一种手段,它是静态的。子类在动态运行时提供了更丰富的多态性。

10.2 C++编程的核心技术

10.2.1 类的定义与使用

1. 类的定义

用户可以创建自己的类,它是用户自己定义的数据类型,组成这种类型的不仅可以是数据(称为类的数据成员),而且可以是对数据进行操作的方法(称为类的函数成员)。属于这种类型的
的具体数据称之为这个类的对象。

(1)类定义的形式

类类型像其他任何数据类型一样,在使用之前必须进行定义。定义类的一般形式为:

```
class
{
    private:
        私有数据与函数
    public;
```

```

        公有数据与函数
    protected;
        受保护的数据与函数
};

```

其中在一对花括号内的部分叫类体；闭花括号后的分号标志类说明语句结束。类内定义的数据和函数分别称为这个类的数据成员与函数成员，类的成员的访问权限，通过关键字来定义，private 表示私有，public 表示公有，protected 表示受保护。

(2) 注意事项

①在类说明中的关键字 private、public 和 protected 出现的顺序可任意，C++ 编译器每遇到有类的权限关键字，就将其后成员的访问权限确定为该关键字所规定的权限。

②数据成员可以具有任何数据类型。由于类是一种数据类型，因而不能在类说明中对数据成员进行初始化。

③在类说明中仅对函数成员进行说明，在程序中还必须定义该函数。定义成员函数的一般形式为：

```

    返回类型 类名::成员函数名(参数说明)
    { 函数体; }

```

其中，“::”为作用域运算符，“类名::”表明其后的函数是属于该类的。

用上述这种形式定义的函数称为外联函数，当然，也可在类的说明中直接写出成员函数的定义，这样的成员函数称为内联函数。

2. 类的使用

(1) 定义类的对象

类是用户定义的一种抽象的数据类型，属于这种类型的具体数据叫做这个类的一个对象，因此，使用类的过程，就是在程序中说明类类型变量的过程，即创建对象的过程。创建对象就是为对象分配内存。在创建对象时，类被用作模板，所以对象又称为类的实例。

定义对象的一般形式为：

```
类名 对象名；
```

(2) 使用对象访问成员函数

定义类的对象之后，就可以访问对象的成员函数。一般形式为：

```
对象名.成员函数名();
```

```
或：对象名.成员函数名(实参表)；
```

其中“.”为取成员运算符，表示访问一个对象的成员。由于代码是被一个类的所有对象所共享的，而数据则是属于各个对象的，因此，当一个对象的成员方法被调用时，可通过类找到成员方法的实现代码，同时通过对象确定需要访问的数据。当然，也可使用类这种类型的指针，并能使用运算符“—>”来访问对象的成员，其形式为：

```
对象名—>成员函数名();
```

```
或：对象名—>成员函数名(实参表)；
```

3. 应用示例

[例 10.7] 输入一个圆的半径，计算圆的面积。

```

#include <iostream.h>
class CircleAreaOOP

```



```

{
    private:
        double radius;
        double area;
    public:
        void setRadius();
        void getArea();
        void printArea();
};

void CircleAreaOOP::setRadius()
{
    cout<<"输入圆的半径: ";
    cin>>radius;
}

void CircleAreaOOP::getArea()
{
    area=3.14159*radius*radius;
}

void CircleAreaOOP::printArea()
{
    cout<<"\n 圆的面积为 "<<area<<endl<<endl;
}

void main()
{
    CircleAreaOOP circle;
    circle.setRadius();
    circle.getArea();
    circle.printArea();
}

```

运行结果:

```

输入圆的半径: 2
圆的面积为 12.5664

```

10.2.2 数据的封装

1. 数据封装的内涵

(1)在C++中,数据封装是通过类实现的。在类内,指定了各个成员的访问权限,用关键字 private 说明的成员是私有的,只有该类的成员函数(还有友员,后面再作介绍)可以对其访问;用关键字 public 说明的成员是公有的,可以被本类之外的函数访问。

(2)若要隐藏数据,需将其数据成员说明为私有的;若将部分成员函数说明为公有的,则为

外界能访问这类的对象提供接口,以使其他函数(如 main()函数)也能访问和处理该类的对象。

(3)对于仅仅是为支持公有函数的实现而不作为对象接口的成员函数,可将它说明为私有的。当数据结构发生变化时,只要修改类的公有成员函数的功能代码,即可保证对象的功能不变,只要对象功能保持不变,则公有成员函数所形成的接口就不会发生变化。这样对对象的内部功能所作的修改就不会影响使用该对象的软件系统。

2. 应用示例

[例 10.8] 输入两个整数,然后进行交换。

```
class SwapTwoNumbers
{
public:
    SwapTwoNumbers();
    void read();
    void swap();
private:
    int *a;
    int *b;
};

SwapTwoNumbers::SwapTwoNumbers()
{
    a=new int; b=new int;
    *a=0; *b=0;
}

void SwapTwoNumbers::read()
{
    cout<<"输入两个整数:";
    cin>>*a>>*b;
}

void SwapTwoNumbers::swap()
{
    cout<<"交换前: a="<<*a<<"    b="<<*b<<endl;;
    *a^=*b;    // 或 *a=*a^*b;
    *b^=*a;    // 或 *b=*a^*b;
    *a^=*b;    // 或 *a=*a^*b;
    cout<<"交换后: a="<<*a<<"    b="<<*b<<endl<<endl;
}

void main()
{
    SwapTwoNumbers obj;
    obj.read();
```

```
    obj.swap();
}
```

运行结果:

输入两个整数: 918 421

交换前: a=918 b=421

交换后: a=421 b=918

10.2.3 函数的重载

1. 函数重载的内涵

函数重载就是一种简单的多态性。在C++中,允许函数重载,函数重载是指函数名相同,而参数的类型、个数、顺序不同,返回类型可以相同可以不同。C++根据该函数的参数类型与个数,而不是根据它的返回类型来区别具有相同名字的重载函数。

2. 应用示例

[例 10.9] 函数重载的使用。

```
#include <iostream. h>
class FunctionOverload
{
    public:
        int square(int x);
        double square(double y);
};
int FunctionOverload::square(int x){ return x * x; }
double FunctionOverload::square(double y){return y * y; }
void main()
{
    FunctionOverload obj;
    cout<<"square(int)的结果:"<<obj.square(7)<<endl;
    cout<<"square(double)的结果:"<<obj.square(7. 5)<<endl<<endl;
}
```

运行结果:

square(int)的结果: 49

square(double)的结果: 56. 25

10.2.4 对象的初始化

1. 初始化的方式

在C++中,对象的初始化可以通过构造函数来实现。构造函数是类的一种特殊的成员函数,它与类同名,即无函数名,且不能有返回类型,其定义形式为:

类名() //无参的构造函数 | 或: 类名()

{		{
函数体		方法体
}		}

当创建对象时,系统就自动地调用构造函数来完成初始化工作,像其他函数一样,构造函数也可以重载。

2. 应用示例

[例 10.10] 求一元二次方程的实根与复根。

```
#include <iostream. h>
#include<math. h>
const int N=4;
class Polynomial
{
    public:
        Polynomial();
        Polynomial(double x, double y, double z);
        void read();
        void roots() const;
    private:
        double a, b, c;
};
Polynomial::Polynomial() { a=b=c=0; }
Polynomial::Polynomial(double x, double y, double z)
{
    a=x; b=y; c=z;
}
void Polynomial::roots() const
{
    double d, x1, x2, re, im;
    if(a==0)
        cout<<"没有根!"<<endl<<endl;
    else
    {
        d=b*b-4*a*c;
        if(d>=0)
        {
            x1=(-b+sqrt(d))/(2*a);
            x2=(-b-sqrt(d))/(2*a);
            cout<<"x1="<<x1<<"    x2="<<x2<<endl<<endl;
        }
    }
}
```

```

        else
        {
            re = -b/(2 * a);
            im = sqrt(-d)/(2 * a);
            cout << "re=" << re << "    im=" << im << endl << endl;
        }
    }
}

void main()
{
    double a, b, c;
    for(int i=1; i<=N; i++)
    {
        cout << "输入三个数: ";
        cin >> a >> b >> c;
        Polynomial value(a, b, c);
        value.roots();
    }
}

```

运行结果:

```

输入三个数: 12 3 -1
x1=0.189576  x2=-0.439576
输入三个数: 12 3 1
re=-0.125  im=-0.260208
输入三个数: 0 0 1
没有根!

```

10.2.5 缺省构造函数、拷贝构造函数与析构函数

1. 缺省构造函数的形式与作用

缺省构造函数是一种特殊的构造函数,它没有参数,而且函数体是空的,形式为:

```
类名() {}
```

在C++中,创建对象总是通过构造函数来完成的,对于没有构造函数的类,系统会自动地给它加上一个缺省构造函数来创建这个类的对象。但由于缺省构造函数的函数体为空,所以创建出来的对象,其状态是不确定的。

2. 拷贝构造函数的形式与作用

拷贝构造函数是另一种特殊的构造函数,它的参数是它所在类的对象的引用,形式为:

```

类名(类名 &对象)
{
    函数体
}

```

```
}
```

当用这个类的一个对象初始化另一个对象时,它被调用,将参数的数据逐个拷贝到新建的对象中。

3. 析构函数的形式与作用

析构函数是与构造函数相反的函数,用于在对象消失时执行一些清理任务的工作,如释放构造函数分配的内存等。

析构函数的形式为:~函数名();

4. 应用示例

[例 10.11] 使用构造函数与析构函数。

```
#include <iostream. h>
#include <string. h>
const int MINID=1;
const int MINScore=0;
const int MAXScore=100;
class Score
{
    public:
        Score(long id1, char * name1, unsigned math1, unsigned physic1);
        void list();
        ~Score();
    private:
        long id;           // 学号
        char name[20];      // 学生姓名
        unsigned math;      // 每门课的成绩
        unsigned physics;
};
inline int min(int x, int y){ return ((x<y)? x:y); }
inline int max(int x, int y){ return ((x>y)? x:y); }
Score::Score(long id1, char * name1, unsigned math1, unsigned physic1)
{
    id=max(MINID, id1); id=min(id, MAXID);
    math=max(MINScore, math1); math=min(math, MAXScore);
    physics=max(MINScore, physic1); physics=min(physics, MAXScore);
    strcpy(name, name1);
}
void Score::list()
{
    cout <<"学号:    " <<id<<endl<<"姓名:    " <<name<<endl
        <<"Math:    " <<math<<endl<<"Physics:    " <<physics
        <<endl<<endl;
```



```

    }
    Score::~~Score() { //just do nothing }
    void main()
    {
        Score score1(124, "王涌", 95, 99);
        score1.list();
        cout<<endl;
        Score score2(918, "程浩", 97, 96);
        score2.list();
        cout<<endl;
    }

```

运行结果:

```

    学号:    124
    姓名:    王涌
    Math:    95
    Physics: 99
    学号:    918
    姓名:    程浩
    Math:    97
    Physics: 96

```

10.3 类成员与对象的构造

10.3.1 使用 this 指针

1. this 指针的用法

(1)在 C++ 中,当一个成员函数被调用时,系统就自动地传递一隐含的参数给成员函数,该隐含参数是一个指向接受该函数调用的对象的指针,这样,成员函数就知道该对哪个对象进行操作。在程序中,可以使用关键字 this 来引用该指针,因而称为 this 指针。

(2)this 指针是 C++ 实现封装的一种机制,它将对象和该对象调用的成员函数连接在一起,因而,从外部来看,每一个对象都拥有自己的函数成员。this 使用形式为:

 this->数据成员=参量;

或:对象指针->指针=this->对象指针;

(3)this 指针在一般情况下不是很有用,不写也没关系,让系统自行设置。然而,在运算符重载、数据结构等情况下,却很有用。例如,在 C 语言中,双向链表是通过结构来实现的,在结构中定义数据项与指针项,可在结构之外再定义函数来实现插入、删除等功能。在 C++ 中,可以通过类来实现链表,把数据项和指针项定义为类的数据成员,同时把对链表的操作定义为类的函数成员,从而实现了数据封装,这样可大大减少出错的可能,如:

```

class mylink
{
    int sum;
    mylink * pre;    //前驱
    mylink * suc;    //后驱
public:
    void append(mylink * p);
    ...
};

```

其中,成员函数 void append(mylink * p)的作用是将 p 指向数据作为新的链表项,插入到当前表项的前面,其实现为:

```

void mylink::append (mylink * p)
{
    p->suc=suc;    // p->suc=this->suc
    p->pre=this;
    suc->pre=p;
    suc=p;
}

```

2. 应用示例

[例 10.12] 计算 $h = \frac{\pi ab}{2c + e^{a-b}}$ 之值。

```

#include <iostream.h>
#include <math.h>
const double PI=3.14159;
class ThisPointer1
{
public:
    ThisPointer1(){};
    void read();
    void result();
    void print();
private:
    double a, b, c, h;
};
void ThisPointer1::read()
{
    cout<<"输入三个数:";
    cin>>a>>b>>c;
}
void ThisPointer1::result()

```

```

{
    h=PI*a*b/(2*c+exp(a-b)); /* 或 this->h=PI*a*b/(2*c+
                                exp(a-b)); */
}
void ThisPointer1::print()
{
    cout<<endl<<"h 的结果是"<<this->h<<endl<<endl; // this->h 或
    h
}
void main()
{
    ThisPointer1 *p;
    p=new ThisPointer1();
    p->read();
    p->result();
    p->print();
    delete p;
}

```

运行结果:

输入三个数: 93 11.19 14.45

h 的结果是 9.65675e-033

10.3.2 使用静态成员

1. 静态成员的作用与含义

在类内,用关键字 `static` 修饰所定义的数据成员或成员函数,统称为静态成员,定义形式为:

```

static 类型 变量名;           // 静态数据成员
static 返回类型 函数名(参数表); // 静态成员函数

```

静态成员为该类的所有对象所共享,或者说它们是属于类的。例如:

```

class A
{
    public;
        static int is;           // 静态数据成员
        static void fun();       // 静态成员函数
}

```

(1)静态成员与一般的成员函数不同,静态成员没有 `this` 指针,除非显式地把指针传给它,否则不能访问对象的数据成员,因为无法确定是对哪个对象的数据进行操作。

(2)由于静态成员函数是属于类的,所以,可以通过“类名::函数名(参数表)”的形式行调用。

(3) 由于静态成员是属于类的, 所以, 一旦把一个数据成员定义成静态的, 则对该类的所有对象, 这个静态成员值在任何时候均为一样的。实际上, 不管该类生成多少个实例, 这些实例在内存中只是共享这惟一的一个静态成员。不管通过哪个对象改变这个数据成员的值, 其他对象的该成员也随之改变。

2. 应用示例

[例 10.13] 使用静态成员。

(1) 使用静态成员函数

```
#include <iostream. h>
#include <string. h>
class Student1
{
    public:
        Student1(char * pName);
        ~Student1();
    protected:
        static Student1 * pFirst;
        Student1 * pNext;
        char name[40];
};
Student1 * Student1::pFirst=0;
Student1::Student1(char * pName)
{
    strcpy(name, pName);
    name[sizeof(name)-1]='\0';
    pNext=pFirst;
    pFirst=this;
}
Student1::~~Student1()
{
    cout<<"this->name"<<endl<<endl;
    if(pFirst==this)
    {
        pFirst=pNext;
        return;
    }
    for(Student1 * ps=pFirst; ps; ps->pNext)
    {
        if(ps->pNext==this)
        {
            ps->pNext=pNext;    //pNext i. e. this->pNext
```

```

        return;
    }
}
Student1 * fn()
{
    Student1 * ps=new Student1("Annie");
    return ps;
}
void main()
{
    Student1 sa("Qing");
    Student1 * sb=fn();
    delete sb;
}

```

运行结果:

Annie

Qing

(2) 使用静态数据成员

```

#include <iostream. h>
#include <string. h>
class Student2
{
    public:
        Student2(char * pName);
        ~Student2();
        static Student2 * FindName(char * pName);
    protected:
        static Student2 * pFirst;
        Student2 * pNext;
        char name[40];
};
Student2 * Student2::pFirst=0;
Student2::Student2(char * pName)
{
    strcpy(name, pName);
    name[sizeof(name)-1]='\0';
    pNext=pFirst;
    pFirst=this;
}

```



```
Student2::~~Student2()
{
    if(pFirst==this)
    {
        pFirst=pNext;
        return;
    }
    for(Student2 * ps=pFirst; ps; ps->pNext)
        if(ps->pNext==this)
        {
            ps->pNext=pNext; //pNext i. e. this->pNext
            return;
        }
}

Student2 * Student2::FindName(char * pName)
{
    for(Student2 * ps=pFirst; ps; ps=ps->pNext)
        if(strcmp(ps->name, pName)==0)
        {
            return ps;
        }
    return(Student2 *)0;
}

void main()
{
    Student2 s1("Qing");
    Student2 s2("Annie");
    Student2 s3("Yong");
    Student2 * ps=Student2::FindName("Annie");
    if(ps)
        cout<<"OK!"<<endl<<endl;
    else
        cout<<"No find!"<<endl<<endl;
}
```

运行结果:

OK!

10.3.3 使用友员

1. 友员的作用与形式

友员是拥有访问类的私有数据成员的普通函数,在类的说明中,只要在函数的前面冠以关键字 friend,即说明该函数为友员,从而可以访问该类对象的私有数据。

定义友员的形式为:friend 返回类型 函数名(参数表);

注意:

- (1)由于友员不是成员函数,用 private 或 public 说明友员,其效果相同;
- (2)引入友员可以避免反复调用类的成员函数,因而提高效率;
- (3)由于友员会破坏数据封装,因而使用友员要小心谨慎,权衡得失。

2. 应用示例

[例 10.14] 使用友员。

```
#include <iostream.h>
class TimeClass
{
    int hours;
    int minutes;
public:
    TimeClass(int new _ hours, int new _ minutes);
    void SetTime(int new _ hours, int new _ minutes);
    void GetTime(int &current _ hours, int &current _ minutes);
    void CopyTime(TimeClass &Time2);
    friend void Time12(TimeClass TheTime);
    friend void Time24(TimeClass TheTime);
};
TimeClass::TimeClass(int new _ hours, int new _ minutes)
{
    hours=new _ hours;
    minutes=new _ minutes;
}
void TimeClass::SetTime(int new _ hours, int new _ minutes)
{
    hours=new _ hours;
    minutes=new _ minutes;
}
void TimeClass::GetTime(int &current _ hours, int &current _ minutes)
{
    current _ hours=hours;
    current _ minutes=minutes;
```

```

    }
    void TimeClass::CopyTime(TimeClass &Time2){ *this=Time2; }
    void Time12(TimeClass TheTime)
    {
        int PMFlag;
        if(TheTime.hours>12)
        {
            TheTime.hours-=12;
            PMFlag=1;
        }
        else
            PMFlag=0;
        cout<<TheTime.hours<<" ";cout<<TheTime.minutes;
        if(PMFlag) cout<<" PM"<<endl;
        else cout<<endl;
    }
    void Time24(TimeClass TheTime)
    {
        cout <<TheTime.hours<<" : "<<TheTime.hours<<" ."
            <<TheTime.minutes<<endl<<endl;
    }
    void main()
    {
        TimeClass Time1(13, 05);
        Time12(Time1);
        Time24(Time1);
    }

```

运行结果:

```

1:5 PM
13:13.5

```

10.3.4 使用对象成员

1. 对象成员的定义

对象成员是在一个类中说明具有类类型的数据成员,其说明形式为:

```

class 类名
{
    类名 1 成员名 1
    类名 2 成员名 2
    ...

```

类名 n 成员名 n

};

(1)要初始化对象成员,必须调用该对象成员所在类的构造函数,其构造函数定义形式为:
类名::类名(参数表 0):成员 1(参数表 1),成员 2(参数表 2),...,成员 n(参数表 n)

{

...//根据参数表 0 初始化其他数据成员

}

其中,参数表 1 提供初始化成员 1 所需的参数,参数表 2 提供初始化的成员 2 所需参数,余类推。这 n 个参数表中的参数来自参数表 0,而初始化类名的非对象成员所需的参数,则由参数表提供。

(2)建立类名的对象时,计算机先调用对象成员的构造函数,初始化对象成员,然后执行类名的构造函数,初始化其他成员。对于同一个类的不同对象成员,计算机根据它们在类中所说明的顺序来确定其构造函数的调用顺序。

(3)析构函数的调用顺序与构造函数相反。

2. 应用示例

[例 10.15] 使用对象成员。

```
#include <iostream.h>
class Object1      // 定义 Object1 类
{
public:
    Object1(int i);
    ~Object1();
private:
    int iVal;
};
Object1::Object1(int i)
{
    iVal=i;
    cout<<"带参的构造函数 object1 " <<iVal<<endl;
}
Object1::~~Object1()
{
    cout<<"析构函数 Object1 " <<iVal<<endl;
}
class Object2      // 定义 Object2 类
{
public:
    Object2(int i, int j, int k);
    ~Object2();
private;
```

```

        Object1 ObjOne;           // Object1 类型的对象
        Object1 ObjTwo;
        int idata;
    };
Object2::Object2(int i, int j, int k):ObjTwo(i), ObjOne(j)
{
    idata=k;
    cout<<"带参构造函数 Object2"<<idata<<" ";
}
Object2::~~Object2()
{
    cout<<"析构函数 Object2"<<" ";
}
/* 初始化对象 Obj,即先初始化对象成员 ObjOne 和 ObjTwo,然后才完成 Obj 本身
   的初始化;在程序结束时,析构函数调用顺序相反 */
void main()
{
    Object2 Obj(10, 25, 2000);
}

```

运行结果:

```

带参的构造函数 Object1 25
带参的构造函数 Object1 10
带参构造函数 Object22000 析构函数 Object2 析构函数 Object1 10
析构函数 Object1 25

```

10.3.5 使用对象数组

1. 对象数组的含义

对象是一个类的类型数据,对象数组就是数组元素由对象组成的数组。

- (1)为了说明一个对象数组,对应的类必须有一个不带参数或带有缺省参数的构造函数;
- (2)对象数组可以是全局或静态的,此时,必须在主程序 main 调用之前对其进行初始化;
- (3)与定义一维数组类似,也可以定义多维对象数组。

2. 应用示例

[例 10.16] 使用对象数组。

```

#include <iostream. h>
class Object
{
public:
    Object();
    Object(int i, float f);

```



```

        int getint();
        float getfloat();
        ~Object();
    private:
        int num;
        float f1;
};
Object::Object()
{
    num=0; f1=0.0;
    cout<<"num="<<num<<" f1="<<f1<<endl;
}
Object::Object(int i, float f)
{
    num=i; f1=f;
    cout<<"num="<<num<<" f1="<<f1<<endl;
}
int Object::getint(){ return num; }
float Object::getfloat(){ return f1; }
Object::~~Object()
{
    cout<<"析构函数被调用."<<endl;
}
void main()
{
    Object a(2000, 10.25);
    cout<<"对象数组 b[2]"<<endl;
    Object b[2];
}

```

运行结果:

```

num=2000 f1=10.25
对象数组 b[2]
num=0 f1=0
num=0 f1=0
析构函数被调用.
析构函数被调用.
析构函数被调用.

```

10.3.6 使用指向对象的指针

1. 对象指针的用法

C++ 支持指向对象的指针,与对象数组一样,只要把类看作一种数据类型,就很容易理解指向对象的指针。

(1)用 new 运算符创建生存期可控的对象,new 返回这个对象的指针。由于类是一种数据类型,所以,使用 new 创建动态对象的语法和创建其他类型的动态变量的情况类似,不同点在于 new 是与构造函数一起起作用的。

(2)当使用 new 创建动态对象时,计算机就首先为这个对象分配所需的内存,然后自动调用构造函数来初始化这块内存,再返回这个动态对象的地址,如果分配内存失败,就给出出错信息。

(3)使用 new 创建的动态对象不用时,必须用 delete 对其删除,以回收这个动态对象所占用的内存,此时,将调用对象所在类的析构函数。

2. 应用示例

[例 10.17] 当 $x=10.25$, $y=13.06$ 时,试求下列式子的值。

$$a = (x - \sqrt{x^2 + 1} + \log|x + \sqrt{x^2 + 1}|)/2, b = \log|y + \sqrt{y^2 + 1}|, c = a + b$$

```
#include <iostream.h>
```

```
#include <math.h>
```

```
class ObjPointer
```

```
{
```

```
    public:
```

```
        ObjPointer(){};
```

```
        void read(double x1, double y1);
```

```
        void result();
```

```
        void print();
```

```
    private:
```

```
        double a, b, c, x, y;
```

```
};
```

```
void ObjPointer::read(double x1, double y1)
```

```
{
```

```
    x=x1; y=y1;
```

```
}
```

```
void ObjPointer::result()
```

```
{
```

```
    a=(x-sqrt(x*x+1)+log(fabs(x+sqrt(x*x+1))))/2;
```

```
    b=log(fabs(y+sqrt(y*y+1)));
```

```
    c=a+b;
```

```
}
```

```
void ObjPointer::print()
```

```

{
    cout<<endl<<"a="<<a<<"  b="<<b<<"  c="<<c
    <<endl<<endl;
}
void main()
{
    ObjPointer *p;
    p=new ObjPointer();
    p->read(10.25, 13.05);
    p->result();
    p->print();
    delete p;
}

```

运行结果:

a=1.48707 b=3.2634 c=4.75047

10.3.7 类型的转换

1. 类型转换的规则

在C++中,类类型对象的类型转换是通过调用构造函数,产生隐藏对象来完成的。类类型的转换是否合法,完全取决于对象所在类的构造函数类型的转换。

2. 应用示例

[例 10.18] 类类型的转换。

```

#include <iostream.h>
class ObjectType
{
public:
    ObjectType(double i);
    void print();
private:
    double data;
};

```

// 系统调用下面构造函数,使 1.24 赋给隐藏对象 data,则 double 型转换为类类型数据

```

ObjectType::ObjectType(double i)
{
    data=i;
    cout<<"初始化值:"<<data<<"  ";
}
void ObjectType::print()
{

```

```

        cout<<"输出结果:"<<data<<endl;
    }
    void main()
    {
        ObjectType N(0); // N 对象为 ObjectType 类型
        cout<<endl;
        N=1.24;          // N 应接受 ObjectType 型数据,但 1.24 是 double 型
        N.print();
        N=ObjectType(10.25); // N 接受 ObjectType 构造函数数据,类型相同
        N.print();
    }

```

运行结果:

初始化值:0

初始化值:1.24 输出结果: 1.24

初始化值:10.25 输出结果: 10.25

10.4 派生类的构造

10.4.1 派生类的定义

1. 定义形式

派生类是从已有的超类创建的新类。新类不仅可以全部继承超类的全部成员,还可以通过下列方式产生新的成员。

- (1)增加新的成员变量;
- (2)增加新的成员函数;
- (3)重新定义已有的成员函数;
- (4)改变现有的成员属性。

定义派生类的一般形式为:

```

class 派生类:访问控制 超类名
{
    private:
        成员说明列表
    public:
        成员说明列表
};

```

其中,派生类设的冒号“:”表示这个新类(派生类)是从超类继承,它继承了超类的所有成员;“访问控制”用于规定超类成员在派生类中的访问权限,即超类成员在派生类中是公有(public)还是私有(private)。

2. 应用示例

[例 10.19] 派生类的使用。

```

#include<iostream. h>
class BaseClass
{
    public:
        void SetValue1(double d1);
        void SetValue2(double d2);
        void ShowValue();
    protected:
        double D1, D2;
};
void BaseClass::SetValue1(double d1){ D1=d1; }
void BaseClass::SetValue2(double d2){ D2=d2; }
void BaseClass::ShowValue()
{
    cout<<"D1="<<D1<<"  D2="<<D2<<endl;
}
class DerivedClass:public BaseClass
{
    public:
        void SetH(double dH);
        void SetW(double dW);
        void ShowHW();
    private:
        double DH, DW;
};
void DerivedClass::SetH(double dH)
{
    DH=dH;
}
void DerivedClass::SetW(double dW)
{
    DW=dW;
}
void DerivedClass::ShowHW()
{
    cout<<"DH="<<DH<<"  DW="<<DW<<endl<<endl;
}
void main()

```



```
{
    DerivedClass data;
    data.SetValue1(11.19);
    data.SetValue2(14.45);
    data.SetH(10.25);
    data.SetW(13.06);
    data.ShowValue();
    data.ShowHW();
}
```

运行结果:

```
D1=11.19  D2 =14.45
DH=10.25  DW =13.06
```

10.4.2 类的保护成员

1. 类的保护成员的定义

在 C++ 中, 派生类的成员函数可以直接访问超类中定义的或超类继承来的公有成员, 但不能访问类的私有成员。然而, 在类定义中, 用关键字 `protected` 说明的则是类的保护成员。受保护的成员具有私有成员和公有成员的双重角色, 对派生类的成员函数来说, 它是公有成员, 可以被访问; 而对其他函数来说则仍是私有成员, 不能被访问。

2. 应用程序

[例 10.20] 类保护成员的使用。

```
#include<iostream.h>
class BaseClass
{
    public:
        void SetValue1(double d1);
        void SetValue2(double d2);
        void ShowValue();
    protected:
        double D1, D2;
};
void BaseClass::SetValue1(double d1){ D1=d1; }
void BaseClass::SetValue2(double d2){ D2=d2; }
void BaseClass::ShowValue()
{
    cout<<"D1="<<D1<<"  D2="<<D2<<endl;
}
class DerivedClass:public BaseClass
{

```

```

    public:
        void SetH(double dH);
        void SetW(double dW);
        void ShowHW();
    private:
        double DH, DW;
};

void DerivedClass::SetH(double dH){ DH=dH; }
void DerivedClass::SetW(double dW){ DW=dW; }
void DerivedClass::ShowHW()
{
    cout <<"D1="<<D1<<"  D2="<<D2<<endl
         <<"DH="<<DH<<"  DW="<<DW<<endl<<endl;
}

void main()
{
    DerivedClass data;
    data.SetValue1(11.19);
    data.SetValue2(14.45);
    data.SetH(10.25);
    data.SetW(13.06);
    data.ShowValue();
    data.ShowHW();
}

```

运行结果:

```

D1=11.19  D2 =14.45
D1=11.19  D2 =14.45
DH=10.25  DW =13.06

```

10.4.3 访问权限的设置

1. 访问权限的含义与使用

访问权限决定着超类各成员在派生类中的访问权限。从访问控制的角度划分,类的派生可分为公有派生与私有派生两种,对于公有派生,其访问控制是 public;而私有派生的访问控制是 private。

(1)在公有派生的情况下,超类成员的访问权限在派生类中保持不变。这意味着:

- ①超类的公有成员在派生类中仍然是公有的;
- ②超类的受保护成员在派生类中仍然是受保护的;
- ③超类的不可访问的和私有的成员在派生类中仍然是不可访问的。

在公有派生的情况下,可以通过定义派生类自己的成员函数来访问派生类对象继承下来

的公有的和受保护的成员,但不能访问继承来的私有成员。当需要类的某些成员能被子类所访问,但又不能被其他的外界函数访问时,就应把它们定义为受保护的,绝不能把它们定义为私有的。否则,在子类中它们就会是不可访问的。

每一个派生类的对象都是超类的一个对象,具体地说,就是下列三种情况(假定 derived 类是从 base 类公有派生的):

①派生类的对象可以赋给超类的对象,如:

```
derived d;  
base b;  
b=d;
```

②派生类的对象可以初始化超类引用,如:

```
derived d;  
base &br=d;
```

③派生类的对象的地址可以赋给指向超类的指针,如:

```
derived &d;  
base &pb=&d;
```

在后两种情况下,通过 pb 或 br 只能访问对象 d 中所继承的超基类成员,这是由超基的定义所决定的。

(2)在私有派生的情况下,通过私有派生,超类的任何成员在派生类中都是私有的。与公有派生一样,超类的私有成员和不可访问成员在派生类中也是不可访问的,而公有成员和受保护的成员,成了派生类的私有成员,派生类的对象想要访问它们,必须定义公有的成员函数作为接口,更重要的是下一次将派生类作为超类进行派生时,它们在新的派生类中将是不可访问的。

2. 应用示例

[例 10.21] 访问权限的使用。

```
#include <iostream.h>  
class BaseClass  
{  
public:  
    double d1, d2;  
    BaseClass() { d1=10.6; d2=14.45; }  
};  
class DerivedClass:private BaseClass  
{  
public:  
    double d3, d4;  
    void show();  
};  
void DerivedClass::show()  
{  
    cout <<"d1=" <<d1<<" d2=" <<d2<<endl
```

```

        <<"d3="<<d3<<" d4="<<d4<<endl<<endl;
    }
    void main()
    {
        DerivedClass data;
        data.d3=9.28;
        data.d4=13.06;
        data.show();
    }

```

运行结果:

```

d1=10.6 d2=14.45
d3=9.28 d4=13.06

```

10.4.4 派生类的构造函数与析构函数

1. 用法

任何类型的对象均有一个初始化问题,派生类也不例外。派生类对象的初始化是通过构造函数进行的。与一般对象不同的是,派生类的对象继承了基类的全部成员,对它们进行初始化时,要调用基类的构造函数,即:

(1)C++中,派生类中继承的超类成员的初始化由派生类的构造函数调用超类的构造函数来实现,这与初始化对象成员是类似之处。假定类B是从类A派生而来,则类B的构造函数一般形式为:

```

B::B(参数表0):A(参数表1)
{
    ... //初始化派生类自己定义的成员
}

```

其中,冒号后面的基类的构造函数的初始化表参数1给出超类的构造函数所需的实参,实参的值可以来自“参数表0”,也可由表达式给出,如果需要超类调用缺省构造函数,则省略这一项。

(2)当创建类B的一个对象时,首先调用超类的构造函数,对超类成员进行初始化,然后再执行派生类的构造函数,初始化对象的派生类的成员。如果某个超类仍然是一个派生类,则这个过程继续进行,当该对象消失时,析构函数的执行顺序和执行构造函数时的顺序相反。

(3)若派生类中包含有对象成员,则对象成员的初始化仍在初始化列表中进行,此时首先调用超类的构造函数,然后调用对象成员的构造函数,最后执行派生类的函数。

2. 应用实例

[例 10.22] 使用派生类的构造函数。

```

#include<iostream.h>
class BaseClass
{
    public:

```

```
        BaseClass(double d);
        ~BaseClass();
    private:
        double dValue;
};
BaseClass::BaseClass(double d)
{
    dValue=d;
    cout<<"构建基类"<<endl;
}
BaseClass::~~BaseClass()
{
    cout<<"撤销基类"<<endl;
}
class DerivedClass:public BaseClass
{
    public:
        DerivedClass(double d1, double d2);
        ~DerivedClass();
    private:
        double dValue1;
};
DerivedClass::DerivedClass(double d1, double d2):BaseClass(d1)
{
    dValue1=d2;
    cout<<"构建派生类"<<endl;
}
DerivedClass::~~DerivedClass()
{
    cout<<"撤销派生类"<<endl;
}
void main()
{
    DerivedClass data(9.18, 4.21);
}
```

运行结果:

```
构建基类
构建派生类
撤销派生类
撤销基类
```


10.4.5 多重继承

1. 多重继承的定义与用法

如果一个派生类只继承一个超类的成员,称之为单一继承;如果一个派生类同时继承了多个超类的情况,则称之为多重继承。在C++中,多重继承的一般形式为:

```
class 派生类名:访问控制 超类名 1, 访问控制 超类名 2, ..., 访问控制 超类名 n
{
    ... //定义派生类自己的成员
};
```

这样定义之后,派生类继承了超类 1 到超类 n 的所有数据成员和成员函数,访问控制用于限制其后超类的成员在派生类中的访问权限,其规则与单一继承一样。多重继承是单一继承的扩展。

2. 应用示例

[例 10.23] 使用多重继承。

```
#include <iostream. h>
class Class1
{
    public:
        void setValue1(double);
        void showValue1();
    private:
        double dValue1;
};
class Class2
{
    public:
        void setValue2(double);
        void showValue2();
    private:
        double dValue2;
};
class Class3:public Class1, private Class2
{
    public:
        void setValue3(double, double);
        void showValue3();
    private:
        double dValue3;
};
```

```
void Class1::setValue1(double d1){ dValue1=d1; }
void Class1::showValue1()
{
    cout<<"dValue1="<<dValue1<<endl;
}
void Class2::setValue2(double d1){ dValue2=d1; }
void Class2::showValue2()
{
    cout<<"dValue2="<<dValue2<<endl;
}
void Class3::setValue3(double d1, double d3)
{
    dValue3=d1;  setValue2(d3);
}
void Class3::showValue3()
{
    showValue2();
    cout<<"dValue3="<<dValue3<<endl;
}
void main()
{
    Class3 data;
    data.setValue1(1993);
    data.showValue1();
    data.setValue3(11.19, 14.45);
    data.showValue3();
    cout<<endl;
}
```

运行结果:

```
dValue1=1993
dValue2=14.45
dValue3=11.19
```

10.4.6 在派生类中改写基类的成员函数

1. 成员函数的改写

一般来说,大部分派生类都是将超类中成员函数全部照搬,但在许多情况下,超类的成员函数是不符合派生类需要的,此时就可以在派生类中改写超类的成员函数。

2. 应用示例

[例 10.24] 在派生类中使用成员函数。

```

#include <iostream.h>
const double PI=3.14159;
class Point
{
    public:
        void setIcon(double d1, double d2);
        double area();
    private:
        double x, y;
};
void Point::setIcon(double d1, double d2)
{
    x=d1; y=d2;
}
double Point::area(){ return 0.0;}
class CircleArea:public Point
{
    public:
        void setRadius(double radius);
        double area();
    private:
        double Radius;
};
void CircleArea::setRadius(double radius){ Radius=radius; }
double CircleArea::area(){ return PI * Radius * Radius; }
void main()
{
    Point p;
    double Area=p.area();
    cout<<"p 点的面积为 "<<Area<<endl;
    CircleArea c;
    c.setRadius(5.10);
    Area=c.area();
    cout<<"C 圆的面积为 "<<Area<<endl<<endl;
}

```

运行结果:

p 点的面积为 0
C 圆的面积为 81.7128

10.4.7 虚拟函数

1. 用法

虚拟函数是一个普通的成员函数,只要在类内说明时,在函数原型前面加上一个关键字 `virtual` 即可。

在一个类内,如果有一个成员函数被说明为成员函数,那么,对于从该类直接或间接派生出去的所有类,只要定义一个与虚拟函数原型相同的函数,且指向基类对象的指针指向派生类对象时,系统就会自动用派生类的同名函数取代虚拟函数。

(1)虚拟函数一旦被说明,就一直具有“虚”的特性,即如果在派生类中没有定义自己的虚拟函数,则它将继承基类的虚拟函数。一般说来,如果类族中某一个类忘记定义自己的虚拟函数,计算机就会从上往下,沿着继承树搜索,将该类的各级基类中最靠近该类的虚拟函数作为它的虚拟函数。

(2)构造函数不能是虚拟函数,但析构函数可以是虚拟函数。

(3)构造函数必须是公有的。

2. 应用示例

[例 10.25] 使用虚拟函数。

```
#include <iostream.h>
const double PI=3.14159;
class Point
{
    public:
        void setIcon(double d1, double d2);
        virtual double area();
    private:
        double x, y;
};
void Point::setIcon(double d1, double d2)
{
    x=d1; y=d2;
}
double Point::area(){ return 0.0;}
class CircleArea:public Point
{
    public:
        void setRadius(double radius);
        double area();
    private:
        double Radius;
};
```

```

void CircleArea::setRadius(double radius){ Radius=radius; }
double CircleArea::area(){ return PI * Radius * Radius; }
void main()
{
    Point p, *pt=&p;
    cout<<"p 点的面积为 "<<pt->area()<<endl;
    CircleArea c;
    c.setRadius(5.10);
    pt=&c;
    cout<<" C 圆的面积为 "<<pt->area()<<endl<<endl;
}

```

运行结果:

p 点的面积为 0

C 圆的面积为 81.7128

10.4.8 纯虚拟函数与抽象类

1. 纯虚拟函数与抽象类的定义

一个虚拟成员函数,如果它的原型设置成等于0,则这个虚拟函数叫纯虚拟函数。这个虚拟函数不是在基类中实现的,它必须在派生类中实现。一个含有纯虚拟函数的类,叫抽象类,抽象类只能从基类派生新的类,不能从抽象类中产生对象。说明纯虚拟函数的形式为:

```

class 类名
{
    virtual 类型 函数名(参数表)=0;
};

```

(1)不能说明抽象类的对象,但可以说明指向抽象类对象的指针变量和引用变量;

(2)抽象类中可以有多个纯虚拟函数;

(3)说明了纯虚拟函数的派生类仍然是抽象类,若派生类中给出了基类所有纯虚拟函数的实现,则这派生类不再是抽象类;

(4)抽象类至少含有一个虚拟函数,而且至少有一个虚拟函数是纯虚拟函数,以便把它与空的虚拟函数区分开来;

例如:

```

virtual void printOn()=0;    // 纯虚拟函数
virtual void printOn(){}    // 空的虚拟函数

```

(5)若派生类中没有重新定义基类中的纯虚拟函数,则在派生类中必须将该虚拟函数说明为虚拟函数。

2. 应用示例

[例 10.26] 使用纯虚拟函数与抽象类。

(1)使用虚拟函数

```
#include<iostream.h>
```



```
class Shape
{
    public:
        void setValue(double d1, double d2);
        virtual void area()=0;
    protected:
        double x, y;
};

void Shape::setValue(double d1, double d2)
{
    x=d1; y=d2;
}

class Triangle:public Shape
{
    public:
        void area();
};

void Triangle::area()
{
    cout<<"三角形面积=1/2 * "<<x<<" * "<<y<<"="
    <<0.5 * x * y<<endl;
}

class Rectangle:public Shape
{
    public:
        void area();
};

void Rectangle::area()
{
    cout<<"矩形面积="<<x<<" * "<<y<<"="
    <<x * y<<endl<<endl;
}

void main()
{
    Shape *p;
    Triangle t;
    Rectangle r;
    p=&t;
    p->setValue(10.25, 13.06);
    p->area();
}
```

```

    p=&r;
    p->setValue(10.25, 13.06);
    p->area();
}

```

运行结果:

三角形面积= $1/2 * 10.25 * 13.06 = 66.9325$

矩形面积= $10.25 * 13.06 = 133.865$

(2) 抽象类(含有虚拟函数与空虚拟函数)

```

#include<iostream.h>
class Class1          // 抽象类
{
    public:
        virtual void Print()=0;
};
class Class2:public Class1    // Class2 类是由 Class1 类派生的空虚拟函数
{
    public:
        virtual void Print();
};
void Class2::Print()
{
    cout<<"派生类 Class2"<<endl;    // 由此派生类可实现抽象类的功能
}
class Class3:public Class2
{
    public:
        virtual void Print();    // Class3 类是由 Class2 类派生的空虚拟函数
};
void Class3::Print()
{
    cout<<"派生类 Class3"<<endl<<endl;
}
void Show(Class1 *c1)
{
    c1->Print();
}
void main()
{
    Class2 *c2=new Class2;
    Class3 *c3=new Class3;
}

```

```

        Show(c2); // Class2::Print();
        Show(c3); // Class3::Print();
    }

```

运行结果:

派生类 Class2

派生类 Class3

10.5 运算符重载

10.5.1 运算符重载的作用与形式

1. 运算符重载的作用与形式

同样的一个运算符,随着所操作的数据不同而具有不同的含义(例如,对于表达式 $5/2$ 和 $5.0/2$,其结果就不同,前者为 2,后者为 2.5)。这就是运算符重载,而且是系统预先定义的运算符重载。

在 C++ 中,任何运算符都是通过函数来实现的,运算符重载其实就是函数重载,当重载一个双目成员运算符函数时,运算符有一个参数,形式为:

类名 operator # (参数表)

其中,#为运算符,operator 为关键字。operator 与一个运算符连用,就构成一个运算符函数名,如 operator+。可以像重载普通函数那样,重载运算符函数 operator+()。

2. 应用示例

[例 10.27] 使用重载运算符+求复数。

```

#include <iostream.h>
class Complex
{
public:
    double x;
    double y;
    Complex(double real=0, double image=0)
    {
        x=real; y=image;
    }
};

Complex operator +(Complex &a, Complex &b)
{
    double real=a.x+b.x;
    double image=a.y+b.y;
    return Complex(real, image);
}

```

```

    }
    void main()
    {
        Complex c1(11.19, 14.45), c2(10.25, 13.06), c3;
        c3=c1+c2; /* 编译器把此表达式解释为函数调用 operator+(c1, c2), 并把返
                    回值赋给 c3 */
        cout<<c3.x<<"+"<<c3.y<<"i"<<endl<<endl;
    }

```

运行结果:

21.44+27.51i

10.5.2 类运算符与友员运算符

1. 类运算符与友员运算符的作用与形式

C++同时具有类运算符和友员运算符。将作为类的成员的运算符函数称为类运算符,而将作为类的友员的运算符称为友员运算符。这两种运算符都能访问该类的私有成员。如果运算符所需的操作数(尤其是第一个操作数)希望进行隐式转换,则该运算符应通过友员来重载。如果一个运算符的操作需要修改类对象的状态,则应使用类运算符,这样更符合封装的要求。同时,=、()、[]、->四个运算符只能用类运算符重载。

类的友员说明形式为:

```
friend 类名 operator 运算符(参数表);
```

2. 应用示例

[例 10.28] 使用友员运算符重载求复数。

```

#include <iostream.h>
class Complex
{
    private:
        double x;
        double y;
    public:
        Complex(double real=0, double image=0)
        {
            x=real; y=image;
        }
        friend Complex operator +(Complex &a, Complex &b);
        void Result();
};
Complex operator +(Complex &a, Complex &b)
{
    double real=a.x+b.x;

```

```

        double image=a.y+b.y;
        return Complex(real, image);
    }
void Complex::Result(){ cout<<x<<"+"<<y<<"i"; }
void main()
{
    Complex c1(11.19, 14.45), c2(10.25, 13.06), c3;
    c3=c1+c2;    /* 编译器把此表达式解释为 operator+(c1, c2), 并把返回值
                  赋给 c3 */

    c1.Result();
    cout<<"+";
    c2.Result();
    cout<<"=";
    c3.Result();
    cout<<endl<<endl;
}

```

运行结果:

$11.19+14.45i+10.25+13.06i=21.44+27.51i$

[例 10.29] 使用类运算符重载求复数。

```

#include <iostream.h>
class Complex
{
    private:
        double x;
        double y;
    public:
        Complex(double real=0, double image=0)
        {
            x=real;
            y=image;
        }
        Complex operator +(Complex &a);
        void Result();
};
Complex Complex::operator +(Complex &a)
{
    double real=a.x+x;
    double image=a.y+y;
    return Complex(real, image);
}

```



```

void Complex::Result() { cout<<x<<"+"<<y<<"i"; }
void main()
{
    Complex c1(11.19, 14.45), c2(10.25, 13.06), c3;
    c3=c1+c2; /* 编译器把此表达式解释为 c1 的成员函数调用 c1.operator+(c2),
               这个调用返回一个临时变量,并把此变量赋给 c3 */
    c1.Result();
    cout<<"+";
    c2.Result();
    cout<<"=";
    c3.Result();
    cout<<endl<<endl;
}

```

运行结果:

11.19+14.45i+10.25+13.06i=21.44+27.51i

10.5.3 ++与--运算符的重载

1. 能被重载的运算符

(1) C++中,允许程序员根据需要为自己定义的类进行运算符重载,除了.、::、*、sizeof、? 五种运算符不能重载外,其他运算符均能重载,表 10.1 示出能被重载的运算符。

表 10.1 能被重载的运算符

+	-	*	/	%	+=	-=	*=	/=	%=
++	--	>	<	>=	<=	!=	==	&&	
&	!	^	>>	<<	&=	!=	^=	>>=	<<=
[]	()	new	Delete	=	+	-			

(2) 增 1 与减 1 运算符可被重载为前缀或后缀运算符,同时也可被重载为类运算符或友员运算符。设@表示++或--,C++编译器对这两个运算符解释如下:

重载方式	表达方式	C++编译器的解释
类运算符	@obj obj@	obj.operator@ () obj.operator@ (0)
友员运算符	@obj obj@	operator@ (obj) operator@ (obj, 0)

当增 1 和减 1 运算符被重载为后缀运算符时,C++编译器将视其等价于表达式:

obj @ 0

因而,调用带有两个参数的运算符函数时,默认向该函数传送一个值为 0 的参数,这个参数的惟一作用就是用来供编译器分辨前后缀,调用时不必显式地使用该参数。

2. 应用示例

[例 10.30] 重载运算符++。

(1)使用重载运算符++

```
#include <iostream.h>
class Increment
{
    public:
        Increment(int iNumber){iData=iNumber;}
        int operator ++();          // 前增 1: ++iData
        int operator ++(int);       // 后增 1: iData++
        void Result();
    private:
        int iData;
};
int Increment::operator ++()
{
    iData++; return iData;
}
int Increment::operator ++(int)
{
    int iNumber=iData; iData++;
    return iNumber;
}
void Increment::Result()
{
    cout<<"Data="<<iData<<endl;
}
void main()
{
    Increment data(510);
    int iNumber=++data; /* 编译器解释为 iNumber=data. ++(); */
    cout<<"iNumber="<<iNumber<<" , ";
    data.Result();
    iNumber=data++; /* 编译器解释为 iNumber=data. ++(0); */
    cout<<"iNumber="<<iNumber<<" , ";
    data.Result();
}
```

运行结果:

iNumber=511, Data=511

iNumber=511, Data=512

(2)用重载友员运算符++

```
#include <iostream.h>
```

```

class Increment
{
    public:
        Increment(int iNumber) {iData=iNumber;}
        friend int operator++(Increment&);           // 前增 1:++data
        friend int operator++(Increment&, int);      // 后增 1:data++
        void Result();
    private:
        int iData;
};

int operator++(Increment& Ref)
{
    Ref.iData++;
    return Ref.iData;
}

int operator++(Increment& Ref, int)
{
    int iNumber=Ref.iData++;
    return iNumber;
}

void Increment::Result()
{
    cout<<"Data="<<iData<<endl;
}

void main()
{
    Increment data(510);
    int iNumber=++data; /* 编译器解释为友员函数调用 iNumber=operator++
                        (data); */
    cout<<"iNumber="<<iNumber<<endl;
    data.Result();
    iNumber=data++; /* 编译器解释为友员函数调用 iNumber=operator++
                    (data, 0); */
    cout<<"iNumber="<<iNumber<<endl;
    data.Result();
}

```

运行结果:

iNumber=511

Data=511

iNumber=511

Data=512

10.5.4 重载 new 和 delete

1. new 和 delete 重载的含义

(1)在 C++ 中,运算符 new 和 delete 用于动态内存管理。为提高程序的运算效率,均需重载 new 和 delete。

(2)new 和 delete 必须被重载为类运算符,在重载时,不管是否显式地指定关键字,C++ 编译器均把它们作为静态成员函数。new 必须返回一个 void * 类型的指针,它的第一个参数必须是 size 类型的参数(在 stddef.h 中定义),而重载 delete 时,它必须定义为返回 void 类型,它的第一个参数为所释放的内存地址,类型必须说明为 void * 类型。

(3)重载 new 运算符可以将原来以字节方式分配内存改为以块为单元分配内存,以提高内存分配效率。

(4)注意

①可以使用 C++ 预定义的 new 创建一个类的动态对象,而使用预定义的运算符 delete 释放这个对象。

②可以直接调用析构函数释放构造函数指针成员所分配的动态内存,即:

```
p->myClass::~~myClass();
```

实际上,在 myclass::operator delete() 中,什么也没有做,仅仅是显示一条信息,当一个类重载了 new 而没有重载 delete 时,有时则需要直接调用析构函数。

③当使用 new 分配一个类的动态数组对象以及删除这个动态数组对象时,都调用 C++ 预定义的 ::operator new() 和 ::operator delete()。

2. 内存分配错误

(1)重载 new 时,若 new 不能按要求完成指定的任务(如内存分配失败),应返回 0。可以用 C++ 函数 set _new _handler() 来创建自己的错误处理器,当全局运算符 new 不能满足内存分配请求时,函数 set _new _handler() 自动被调用。

(2)注意

①程序员定义的转换函数和类运算符均可由派生类继承,在派生类中定义的转换函数和运算符函数将隐藏基类中定义的这些函数。

②new 和 delete 只能重载为静态函数,不能说明为虚拟函数。成员函数 operator=() 既不能被派生类继承,也不能说明为虚拟函数。其他类运算符和转换函数可以说明为虚拟函数,这种运算符虚拟函数在派生类重定义要求和调用中发生的多态性行为 and 一般的虚拟函数一样。

③友元运算符类体系中的情况和友元函数的情况一样。

10.6 输入/输出流的使用

在 10.1 曾简单地介绍了 C++ 的输入/输出,这里进一步介绍 C++ 输入/输出流的使用。

10.6.1 标准的屏幕输出

1. 使用 cout 流和插入操作符<<

(1)用法

使用 cout 和插入操作符<<可以把显示文本送到计算机的显示器上,使用形式为:

```
cout{<<变量名|<<"字符串"|endl}1+;
```

(2)注意事项

①由于 cout 定义在 iostream.h 文件中,因此,必须在程序开头写上:

```
#include <iostream.h> 或; #include "iostream.h"
```

②在一个 C++ 输出语句中,可以串联多个插入操作符<<来输入多个数据项。

③插入操作符可以处理所有标准的 C++ 类型,如 char、short、int、long、char * (字符串指针)、float、double、void *。

④endl 符号使得输出流移到下一行首,可以把 endl 放在流的任何地方。

(3)应用示例

[例 10.31] 使用 cout 和<< 实现输出。

```
#include<iostream.h>
void main()
{
    char ch='A';
    int i=2000;
    double d=13.06;
    char s[30]="Hello, Annie";
    cout<<"ch:  "<<ch<<" ";
    cout<<"i:  "<<i<<" ";
    cout<<"d:  "<<d<<" ";
    cout<<"s:  "<<s<<endl;
}
```

运行结果:

```
ch:  A, i:  2000, d:  13.06, s:  Hello, Annie
```

2. 调用流成员函数

(1)用法

①使用成员函数 put()可以输出单个字符到流中,使用形式为:

```
char ch;
...
cout.put(ch);
```

②使用成员函数 put()也可输出字符串,使用形式为:

```
char s[]; //字符数组说明
...
cout.put(s[i]);
```


③使用成员函数 `write()` 可以输出一串字符, 使用形式为:

```
cout.write(字符数组名,sizeof(字符数组名));
```

其中, `write` 的第二个参数说明了从第一个参数中复制字符的个数。`write` 复制指定数目的字符, 包括空字符。例如:

```
char prompt[]="please enter your name:";
cout.write(prompt,sizeof(prompt));
```

(2) 注意事项

`put()` 和 `write()` 成员函数实际上都是返回一个对流对象的引用。这里给出的例子忽略了返回值, 通常使用这个返回值来测试错误状态。

(3) 应用示例

[例 10.32] 使用流成员函数实现输出。

```
#include<iostream.h>
void main()
{
    char ch='a';
    char s[30]="Hello, Annie";
    cout<<"ch:   ";
    cout.put(ch)<<" ";
    cout<<"s:   ";
    cout.write(s, 13)<<" ";
    cout<<"s[7]: ";
    cout.put(s[7])<<endl;
}
```

运行结果:

```
ch:   a, s:   Hello, Annie, s[7]:   A
```

3. 使用格式化输出

(1) 使用操作器

①用法

由于仅有少量的流操作器定义在 `iostream.h` 文件中, 大部分流操作器定义在 `iomanip.h` 文件中, 因此, 编写程序时, 必须在程序开头写上 `#include <iomanip.h>`, 才能获得对所有操作器的访问权。使用操作器时, 要把它放在插入操作符的串联序列中。一般形式为:

```
cout<<{endl|dec|hex|oct|setbase(int n)|setfill(int c)
|setprecision(int n)|setw(int n)|ends}1<<变量名{<<endl}opt;
```

注意, 大部分操作器既可用于输入流, 也可用于输出流, 当在输入流中使用时, 它将控制输入的处理过程, 下面以输出为例具体说明操作器用法:

- `dec` 转换基数为十进制形式

```
cout<<dec<<sum<<endl; 使得 sum 值以十进制显示。
```

注意: C++ 流中缺省的转换数制是十进制; 当转换成其他数制后, 再重新选择十进制时, 必须使用 `dec` 操作器。

- `hex` 转换基数为 16 进制形式

`cout<<hex<<sum<<endl;` 如果 `sum` 的十进数值是 255, 则使用此语句后将得到 `ff`(16 进制数, 表示十进制数 255)。

在一个语句中, 可通过在不同位置上插入适当的基数选择操作器, 实现混合的格式输出, 如 `cout<<"Base 10:"<<dec<<sum<<"Base 16:"<<hex<<sum<<endl;`

- `oct` 转换基数为八进制形式。

- `setbase(int n)` 设置所需的基数, 例如八进制、十进制或十六进制, 它的使用与 `oct`、`dec`、`hex` 操作器同。

- `setw(int n)` 设置输出域为 `n` 个字符宽, 用于输入时, 仅适用于字符输入。

- `setprecision(int n)` 设置浮点数输出精度, 即设置浮点数为 `n` 位(含小数点)的浮点数, 缺省值为 6 位。

- `setfill(int c)` 设置填充字符, 它通常与 `setw()` 配合使用。当一个项目输出未达到预定的宽度时, 以空字符填充。注意, 输出流的缺省是右对齐, 如:

```
cout<<setw(10)<<setfill(' *')<<918<<endl;
```

输出结果为: `***** 918`

- `endl` 在流中插入一个换行符, 并刷新流。

- `ends` 把空终结符(`'\0'`)插入到串中。

- `flush` 刷新与流相联系的缓冲区。

- `ws` 仅用于输入流中, 用于跳过空字符。

②应用示例

[例 10.33] 使用操作器实现格式化输出。

```
#include<iostream. h>
#include<iomanip. h>
void main()
{
    int i=1119;
    double d=2.451993;
    cout<<dec<<"dec: "<<i<<" ";
    cout<<hex<<"hex: "<<i<<" ";
    cout<<oct<<"oct: "<<i<<" ";
    cout<<dec<<"base 10: "<<i<<endl;
    cout<<setprecision(3)<<"d:      "<<d<<" ";
    cout<<"d:      "<<setw(15)<<setfill(' *')<<d<<endl<<endl;
}
```

运行结果:

```
dec:  1119, hex:  45f, oct:  2137, base 10:  1119
```

```
d:           2.45, d:           ***** 2.45
```

(2)使用流的格式化标志

①用 `setiosflags()` 和 `restiosflages()` 操作器

a)使用 `setiosflags()` 设置格式标志

`setiosflags()` 操作器定义为: `setiosflags(flag f)`

其中, f 为设置的格式标志位, 具体选项为表 10.2 所给出的常量, 使用时必须用 `ios::` 作为每个格式标志位的开头, 使用形式为:

```
cout<<setiosflags(ios::格式标志位)<<变量名{<<endl }opt;
```

例如, 选定用定点形式表示浮点数, 则为:

```
float PI=3.14159;
```

```
cout<<setiosflags(ios::fixed)<<PI<<endl;
```

此时输出为 3.14159

若选定用科学记数法, 则为 `cout<<setiosflags(ios::scientific)<<PI<<endl;`

此时输出为 3.14159e+0.0。

使用或运算符可以在一个操作中使用多个标志位, 使用形式为:

```
cout<<setiosflags(ios::格式标志位|ios::格式标志位)opt<<变量名<<{<<endl }opt; 如:
```

```
cout<<setiosflags(ios::dec|ios::showbase)<<sum<<endl;
```

表 10.2 ios 标志位

标志位	含义
skipws	跳过输入中的空白符
left	数据左对齐输出
right	数据右对齐输出
internal	数符左对齐, 数值右对齐, 符号和数值之间为填充符
dec	转换基数为十进制形式
oct	转换基数为八进制形式
hex	转换基数为十六进制形式
showbase	输出一个带有基指示符的数值
showpoint	浮点数输出, 而且总是带有小数点
uppercase	用大写形式输出十六进制数
showpos	在正数前加一个“+”号
scientific	用科学记数法表示浮点数
fixed	用定点数形式表示浮点数
unitbuf	完成插入操作后, 立即刷新流的缓冲区
stdio	完成插入操作后, 刷新系统的 stdout stderr

b) 使用 `restiosflags()` 清除标志位设置

使用形式为: `cout<<restiosflags(ios::格式标志位)<<变量名{<<endl }opt;`

例如, 清除所设置的 `showbase`: `cout<<resetiosflags(ios::showbase)<<sum<<endl;`

c) 应用示例

[例 10.34] 使用流格式化标志实现格式化输出。

```
#include<iostream.h>
#include<iomanip.h>
void main()
{
    int i=124;
    cout<<setiosflags(ios::left);
```

```

cout<<"Base 10:  " <<setw(10)<<setfill(' * ')<<i<<endl;
cout<<setiosflags(ios::left);
cout<<setiosflags(ios::hex);
cout<<"Base 16:  " <<setw(10)<<setfill(' * ')<<i<<endl;
cout<<resetiosflags(ios::hex);
cout<<setiosflags(ios::left|ios::oct);
cout<<" Base 8:  " <<i<<endl;
cout<<resetiosflags(ios::oct);
cout<<setiosflags(ios::dec|ios::showpos);
cout<<"Base 10:  " <<i<<endl;
cout<<setiosflags(ios::showbase|ios::hex);
cout<<"Base 16:  " <<i<<endl;
cout<<resetiosflags(ios::hex);
cout<<setiosflags(ios::showbase|ios::oct);
cout<<" Base 8:  " <<i<<endl<<endl;
}

```

运行结果:

```

Base 10:  124 *****
Base 16:  7c *****
Base 8:   174
Base 10:  +124
Base 16:  0x7c
Base 8:   0174

```

②使用 setf()和 unsetf()成员函数

a)用法

setf()用于设置标志位,unsetf()用于清除标志位,这两个函数的功能与 setiosflags()和 resetiosflags()相似。然而,setf()和 unsetf()是真正的成员函数,可以直接访问它们,使用形式为:

cout.setf(ios::格式标志位);		cout.setf(ios::scientific);
cout.unsetf(ios::格式标志位);		cout.unsetf(ios::sicientific);

b)应用示例

[例 10.35] 使用 setf()和 unsetf()成员函数实现输出。

```

#include<iostream.h>
#include<iomanip.h>
void main()
{
    double d=2000.102512;
    cout<<"d:  " <<d<<endl;
    cout<<setw(15);

```

```

    cout<<setiosflags(ios::left|ios::fixed);
    cout<<setfill(' *')<<"fixed:  ";
    cout<<d<<endl;
    cout<<resetiosflags(ios::fixed);
    cout<<setiosflags(ios::right|ios::scientific);
    cout<<"scientific:  " <<d<<endl;
    cout.unsetf(ios::scientific);
    cout.setf(ios::fixed);
    cout<<"fixed:          " <<d<<endl<<endl;
}

```

运行结果:

```

d:  2000.1
fixed:  ***** 2000.102512
scientific:  2.000103e+003
fixed:  2000.102512

```

(3) 使用格式输出的成员函数

① 用法

输入输出流有 6 个成员函数可供用户调整格式化特征,其中有些成员函数具有与操作器完全一样的特征。

• fill(char)和 fill()

fill(char)用于设置填充字符,使用形式为:

```
cout.fill(字符);      |      cout.fill(' *');
```

不带参数的 fill()用于获取当前的填充字符,使用形式为:

```
当前填充字符=cout.fill();      |      fillchar=cout.fill();
```

• precision(int)和 precision()

precision(int)用于设置浮点数的精度(位数从十进制小数点右边开始计算),使用形式为:

```
cout.precision(位数);      |      cout.precision(3); //设置 3 位精度的十进制浮点数
```

注意:带参数的 precision(int)和 setprecision()操作器完全相同。不带参的 precision()用于获取当前精度。

• width(int)和 width()

width(int)用于设置域宽(在设置域宽时,width(int)与 setw()操作器的作用相同)。使用形式为:

```
cout.width(域宽);
```

不带参的 width()用于获取当前的域宽。

② 应用示例

[例 10.36] 使用格式输出的成员函数实现输出。

```

#include<iostream.h>
#include<iomanip.h>
void main()
{

```



```

double d1=1.24, d2=12.25, d3=520.7, d4=68.172068;
cout.width(10);
cout.fill(' ');
cout<<d1<<" ";
cout.width(10);
cout.fill(' ');
cout<<d2<<endl;
cout.width(10);
cout.fill(' ');
cout<<d3<<" ";
cout.width(10);
cout.fill(' ');
cout.precision(7);
cout<<d4<<endl;
}

```

运行结果：

```

***** 1.24      ***** 12.25
***** 520.7     ** 68.17207

```

10.6.2 标准的键盘输入

1. 使用 cin 流和提取操作符 >> 输入数据

(1) 用法

使用 cin 和 >>, 它们从流中得到数据, 并将这些数据送到变量中, 使用形式为:

```
cin>>变量名{>>变量名}o+;
```

(2) 应用示例

[例 10.37] 使用 cin 和 >> 实现数据输入。

```

#include <iostream.h>
void main()
{
    char ch;
    int i;
    float f;
    double d;
    char s[60];
    cout<<"输入一个字符:";
    cin>>ch;
    cout<<"ch="<<ch<<endl;
    cout<<"输入一个整数:";
    cin>>i;
}

```

```

    cout<<"i="<<i<<endl;
    cout<<"输入一个浮点数:";
    cin>>f;
    cout<<"f="<<f<<endl;
    cout<<"输入一个字符串:";
    cin>>s;
    cout.write(s, 9)<<endl;
}

```

运行结果:

输入一个字符: A

Ch=A

输入一个整数: 2000

i=2000

输入一个浮点数: 10.25

f=10.25

输入一个字符串: Liu-Annie

2. 使用成员函数输入文本与字符

(1) 用法

① 输入单个字符(get(char&))

get()成员函数用于从输入流中读入一个字符,并把它放到 char 变量中传给 get()函数,使用形式为:

char ch;	或:	char ch;
...		...
cin.get(ch);		ch=cin.get();

② 输入字符串(getline())

```
getline(char * buffer, int, char='\n');
```

其中,int 参数说明该输入能处理的最多字符个数(这可防止字符数组溢出);最后一个 char 参数可省略,其缺省值的限制符是换行符。getline()读入所有的文本,直到遇到这个限制符为止。因此,一般输入文本的形式为:

```
cin.getline(数组名,sizeof(数组名));
```

如: char Inputline[80];

```
cin.getline(Inputline, sizeof(Inputline));
```

或使用下列文本输入形式:

```
char Inputline[80];
```

```
cin.read(Inputline, n);
```

其中,n 为输入的字符个数。

③ 忽略位于输入流中的空白符

eatwhite()函数用于忽略位于输入流中的空白符,使用形式为:

```
cin.eatwhite();
```

④忽略和放弃实际输入的字符

使用 `ignore(n)` 能跳过 `n` 个字符, `n` 的缺省值为 1, 因此 `cin.ignore()`; 将跳过下一个输入字符, 当设置 `n` 值大于 1 时, 该函数遇到限制符为止。

(2)应用示例

[例 10.38] 使用成员函数实现文本与字符输入。

```
#include <iostream.h>
const int N=12;
void main()
{
    char ch, s[N];
    cout<<"输入一个字符:";
    ch=cin.get(); //或用 cin.get(ch)
    cout<<"输入的字符是:"<<ch<<endl;
    cin.ignore();
    cout<<"输入一个字符串(输入 N 个字符,然后按 Enter 键):";
    cin.get(s, N);    /* 或用 cin.getline(s, N);、cin.getline(s, N, '$');、
                        cin.read(s, 10); */
    cout<<"输入的字符串是:";
    cin.ignore();
}
```

运行结果:

输入一个字符: e

输入的字符是: e

输入一个字符串(输入 N 个字符,然后按 Enter 键):E-business

输入的字符串是: E-business

10.6.3 用户自定义类的输入/输出

1. 用法

通过对运算符 `>>` 和 `<<` 进行重载, 就可以对自定义的类类型的数据进行输入/输出操作。

在 C++ 中, `<<` 和 `>>` 重载有一定的形式, 对 `<<` 的重载形式为:

```
ostream & operator <<(ostream & stream, CLASS a)
{
    <对于类 CLASS 的输出操作>
    return stream;
}
```

其中, `<<` 运算符重载函数有两个参数, 第一个是 `ostream` 类的一个引用, 第二个是自定义的类类型 `CLASS` 的一个对象。可见, 这个重载是友员的重载。这个函数的返回类型是 `ostream` 类型的引用, 实际上, 在函数中返回的是该函数的第一个参数, 目的是使 `<<` 能连续使用。如:

cout<<a<<b<<c; //a、b、c 均为自定义类型的对象

首先,系统把 cout<<a 作为:

```
ostream & operator <<(cout, a);
```

来处理,返回 cout,紧接着又将刚返回的 cout 连同后面的<<b 一起作为:

```
ostream & operator<<(cout, b);
```

来处理,再返回 cout,然后再处理 cout<<c,从而实现运算符<<的连续使用。

为了输出到 cout 流,引用 ostream 流类,为了从 cin 输入,应引用 istream 流类,因为 cout 是从 ostream 类间接派生出来的,cin 是从 istream 类间接派生出来的。

2. 应用示例

[例 10.39] 使用用户自定义的类实现输入/输出。

```
#include <iostream.h>
#include <string.h>
#include <stdio.h>
typedef char string80[80];
class CDate
{
public:
    CDate(int MonthNum, int DayNum, int YearNum);
    void SetDate(int MonthNum, int DayNum, int YearNum);
    void GetDate(int &MonthNum, int &DayNum, int &YearNum);
    void GetStringDate(string80 &DateStr);
    void GetMonth(string80 &DateName);
    friend ostream &operator<<(ostream &stream, CDate &ADate);
    friend istream &operator>>(istream &stream, CDate &ADate);
protected:
    int Month, Day, Year;
};
ostream &operator<<(ostream &stream, CDate &ADate)
{
    stream<<ADate.Month<<"/"<<ADate.Day<<"/"
    <<ADate.Year<<endl;
    return stream;
}
istream &operator>>(istream &stream, CDate &ADate)
{
    stream>>ADate.Month>>ADate.Day>>ADate.Year;
    return stream;
}
CDate::CDate(int MonthNum, int DayNum, int YearNum)
{
```

```

        SetDate(MonthNum, DayNum, YearNum);
    }
void CDate::SetDate(int MonthNum, int DayNum, int YearNum)
{
    Month=MonthNum;
    Day=DayNum;   Year=YearNum;
}
void CDate::GetDate(int &MonthNum, int &DayNum, int &YearNum)
{
    MonthNum=Month;
    DayNum=Day;   YearNum=Year;
}
void CDate::GetStringDate(string80 &DateStr)
{
    sprintf(DateStr, "%d-%d-%d", Month, Day, Year%100);
}
void CDate::GetMonth(string80 &MonthName)
{
    static char *Months[]={"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                           "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
    strcpy(MonthName, Months[Month-1]);
}
void main(void)
{
    CDate USDate(11, 19, 1993);
    cout<<"当前日期:"<<USDate<<endl;
    cout<<"输入现在日期:";
    cin>>USDate;
    cout<<endl<<"你输入的日期是"<<USDate<<endl;
}

```

运行结果:

当前日期: 11/19/1993

输入现在日期: 10 6 1993

你输入的日期是 10/6/1993

10.6.4 格式化字符串流类的使用

1. 用法

C++提供了两个字符串流类 `ostrstream` 和 `istrstream`。`ostrstream` 是从 `ostream` 派生出来的,它用来把串联间接表中说明的项格式化成字符串。`istrstream` 是从 `istream` 中派生出来的,

它用于把文本项转换成变量所需的内存格式。这两个流都在 `strstream.h` 文件中。因此,在使用字符串流的程序中要写上头文件 `strstream.h`。

注意, `ostrstream` 的使用与 C++ 中其他的流类型相同。如果为支持新的数据类型而重载插入操作符 `<<`, 则也可以把这个新的数据类型用于 `ostrstream` 输出。这是由于 `ostrstream` 是从 `ostream` 中派生出来的, 它继承了 `ostream` 的所有能力, 包括向 `ostream` 增加的能力。

2. 应用示例

[例 10.40] 使用 `ostrstream` 实现数据输出。

```
#include <iostream.h>
#include <strstream.h>
void main()
{
    int iData=3795;
    char buffer[40];
    ostrstream os(buffer, 40);
    os<<iData<<ends;
    cout<<"Buffer="<<buffer<<endl;
    double dData=10.25;
    os.setf(ios::fixed|ios::showpoint);
    os.seekp(0);
    os<<"dData 的值="<<dData<<ends;
    cout<<buffer<<endl<<endl;;
}
```

运行结果:

Buffer=3795

dData 的值=10.250000

[例 10.41] 使用 `istrstream` 实现数据输入。

```
#include <iostream.h>
#include <strstream.h>
void main()
{
    int iData;
    double dData;
    char buffer[40];
    cout<<"输入一个整型数和浮点数:";
    cin.getline(buffer, sizeof(buffer));
    istrstream is(buffer, 40);
    is>>iData>>dData;
    cout<<"你输入的数是:"<<iData<<" 和 "<<dData<<endl<<endl;
}
```

运行结果:

输入一个整型数和浮点数: 3795 10.25

你输入的数是: 3795 和 10.25

10.6.5 磁盘文件的输入/输出

1. 向文件写文本输出

(1) 用法

① 打开一个文件

在使用文件流之前,需要先说明一个 `fstream`(`filestream` 的缩写)类型的对象,`fstream` 适合于所有的文件类型(包括文本文件与二进制文件)的输入/输出,它定义在 `fstream` 文件中,打开一个文件有两种方法:

方法一:用 `fstream` 说明一个文件后,用 `open()` 成员函数打开这个文件,形式为:

`fstream` 流文件名;

流文件名.`open`("要打开的文件名", 文件的访问方式);

如: `fstream outfile`;

`outfile.open("test.dat", ios::out);`

其中,`open()` 的第一个参数为要访问的文件名,可以包含驱动器符号的全部路径,如:

`outfile.open("c:\logs\test.dat", ios::out);`

第二个参数为文件的访问方式,其包括读写,读/写以及二进制数据模式,表 10.3 给出了文件访问方式常量,可以用或操作符把表中给出的常量组在一起,以选择多种特性,例如:`ios::in | ios::out | ios::binary` 表示用二进制读写访问方式打开文件,在程序中可以对该文件进行方块字写操作。

表 10.3 ios::文件访问常量

格式名	用途
<code>in</code>	以输入方法(读方式)打开文件
<code>out</code>	以输出方式(写方式)打开文件
<code>app</code>	以输出方式打开文件,以便向文件的末尾添加新的数据
<code>ate</code>	文件打开时,该文件指针定位于文件末尾
<code>trunc</code>	若文件已存在,将其长度截断为 0,并清除原有内容,若文件不存在,则创建一个新文件
<code>binary</code>	以二进制方式打开文件
<code>nocreate</code>	打开一个已有的文件,若该文件不存在,则打开失败
<code>Noreplace</code>	若文件已存在,则打开失败
<code>ios::out ios::binary</code>	以二进制写方式打开文件
<code>ios::in ios::binary</code>	以二进制读方式打开文件

方法二:把文件名和访问方式作为文件标识符说明的一部分,形式为:

`stream.流文件名`("要访问的文件名", 访问方式);

如: `fstream outfile("test.dat", ios::out);`

此语句说明创建一个 `outfile` 流,用写方式打开文件,并把流与文件联系起来。

② 向已打开的文件写数据

使用标准的流输出语句,可以向已打开的文件写数据。如:

```
outfile<<"Table="<<Table<<endl;
```

③关闭文件

当结束一个文件使用权时,应调用 close()成员函数关闭这个文件,形式为:

```
流文件名.close();
```

如: outfile.close();

(2)应用示例

[例 10.42] 把文本写入磁盘文件。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void main(void)
{
    fstream outfile;
    outfile.open("test.dat", ios::out);
    if(!outfile)
    {
        cout<<"打开 test.dat 文件出错"<<endl;
        abort();
    }
    outfile<<"Componentware Method and Neural Network."<<endl;
    outfile<<"CASE:Computer-Aided Software Engineering."<<endl;
    outfile<<"Automating Software Design "
        <<"and Intentional Programming."<<endl;
    outfile.close();
}
```

2. 从文件中读文本输入

(1)从文本文件读文本数据

①用法

a)使用流系统能从文件内读文本,要以读方式打开文件,应使用 ios::in 文件方式。

b)为了从文件中读文本,应使用 getline()成员函数读入每个输入行。

c)当从文件中读文本时,程序必须检测文件的结束状态。通过检查 eof(),成员函数的返回值来测试文件的结束状态(若不检查文件的结束状态,则当试图读过文件尾时,会出现错误),如:

```
char textline[80]
while (!infile.eof())                //重复读文件,直到文件结束
{
    infile.getline(textline, sizeof(textline)); //从文件读一行文本
    cout<< textline<< endl;           //显示文本行到屏幕上
}
```

②应用示例

[例 10.43] 从文本文件读文本数据。

```
#include <iostream. h>
#include <fstream. h>
#include <stdlib. h>
void main(void)
{
    fstream infile;
    infile. open("test. dat", ios::in);
    if(!infile)
    {
        cout<<"打开 test. dat 文件出错."<<endl;
        abort();
    }
    char textline[80];
    while(!infile. eof())
    {
        infile. getline(textline, sizeof(textline));
        cout<<textline<<endl;
    }
    infile. close();
}
```

运行结果:

// 输出 test. dat 文件的内容

3. get()和 put()成员函数的使用

为加快对文件的访问速度,可使用成员函数 get()和 put()来输入/输出单个字符。

(1)使用 put()成员函数向文件输出字符数据

①用法

调用 put()成员函数输出一个字符,使用 for 语句每次向文件输出一个字符。

```
...
fstream outfile;
Outfile. open("文件名. dat", ios::out)
...
char textline[]={"字符串"};
for(int i=0;i<strlen(textline);i++)
    outfile. put(textline[i]);
...
```

②应用示例

[例 10.44] 使用 put()成员函数把文本写入磁盘文件。

```
#include <iostream. h>
```

```
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
    fstream outfile;
    outfile.open("testp.dat", ios::out);
    if(!outfile)
    {
        cout<<"打开 testp.dat 文件出错."<<endl;
        abort();
    }
    char textline[] =
        { "Componentware Method and Neural Network. \n"
          "CASE;Computer-Aided Software Engineering. \n"
          "Automating Software Design and Intentional Programming. "
        };
    for(unsigned int i=0;i<=strlen(textline);i++)
        outfile.put(textline[i]);
    outfile.close();
}
```

(2)使用 get()成员函数从一个文件中读入字符数据

①用法

使用 get()成员函数能一个字符一个字符地读磁盘文件,只要没有错误或没有达到文件末尾,get()函数总是返回真。因而需要在 get()读入下一个字符的同时检测读的结果。这些均在一个语句中完成。每次从文件中读入一个字符,并使其出现在屏幕上,如:

```
...
fstream infile;
...
char ch;
while(infile.get(ch))
    cout<<ch;
...
```

②应用示例

[例 10.45] 使用 get()成员函数从磁盘文件读文本数据。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void main(void)
{
```



```

    fstream infile;
    infile.open("test.dat", ios::in);
    if(!infile)
    {
        cout<<"打开 testp.dat 文件出错"<<endl;
        abort();
    }
    char ch;
    while(infile.get(ch))
    {
        cout<<ch;
    }
    infile.close();
    cout<<endl;
}

```

运行结果:

// 输出 test.dat 文件的内容

4. 使用 get()和 put()成员函数拷贝整个文件

(1)用法

把 get()和 put()成员函数结合在一起,就能实现把一个文件拷贝到另一个文件,方法是,打开一个已有的文件(即源文件,从这个文件拷贝),然后打开一个新文件(即目标文件,拷贝到这个文件),再把 get()和 put()的调用结合在一起,即可实现文件拷贝,如:

```

...
void main()
{
    fstream infile;
    fstream outfile;
    infile.open("源文件名.dat", ios::in);
    outfile.open("拷贝的目的文件", ios::out);
    ...
    char ch;
    while(infile.get(ch))
        outfile.put(ch);
    infile.close();
    outfile.close();
}

```

(2)应用示例

[例 10.46] 使用 get()和 put()拷贝整个文件。

```

#include <iostream.h>
#include <fstream.h>

```

```

#include <stdlib.h>
void main(void)
{
    fstream infile;
    fstream outfile;
    infile.open("test.dat", ios::in);
    if(!infile)
    {
        cout<<"打开 test2.dat 文件出错."<<endl;
        abort();
    }
    outfile.open("test2.dat", ios::out);
    if(!outfile)
    {
        cout<<"打开 test2.dat 文件出错."<<endl;
        abort();
    }
    char ch;
    while(infile.get(ch))
        outfile.put(ch);
    infile.close();
    outfile.close();
}

```

5. 使用二进制数据文件

(1) 用法

① 把二进制数据输出到流中

a) 为把二进制数据输出到流中, 使用 `write()` 成员函数。该函数有两个参数, 第一个参数是一个指向字符数组的指针, 第二个参数是从数组写到文件的字节个数, 由于 `write()` 要求一个指向字符数组的指针, 所以可把它作为一个高速文本输出子程序使用。例如:

```

char textline[80];
...
outfile.write(textline, strlen(textline));

```

这个语句输出存储在 `textfile` 中的数据, `write()` 总是按指定的字节数输出, 不管数组中是否包含空终结符, 即使 `write()` 检测到缓冲区中的空终结符, 它仍然继续把指定的数据缓冲区的字符写到文件中。事实上, `write()` 可以越过数据缓冲区中的空终结符, 直到把需要的字节数从缓冲区写到文件中为止。

b) 若要写任意数目的二进制数据, 则需要一些额外信息, 因为 `write()` 函数需要一个 `char *` 参数, 所以必须把要写的数据转换为 `char *` 类型, 例如, 给定的数据纪录为:

```

struct person _info
{

```

```

    char name[20];
    float height;
    unsigned short age;
} Aperson;

```

用 write() 成员函数可以写 Aperson 结构:

```
outfile.write((char *) &Aperson, sizeof(Aperson));
```

其中, 转换操作把 Aperson 的地址转换为 char * 类型, sizeof 指定向文件写的字节个数, 给定这些参数后, write() 从 Aperson 中拷贝指定的字节个数到输出文件。

②把二进制数据输入到流中

为把任意数目的二进制数据输入到流中, 即从文件读出数据, 则要使用 read() 函数, 其用法与 write() 类似, 例如, 给定数据记录为:

```

struct person _info
{
    char name[20];
    float height;
    unsigned short age;
};
person _info people[4] = { "Qing", 1.6, 28, "Annie", 0.67, 1,
                           "Yong", 1.8, 33, "Hao", 1.82, 29 };

```

则可用 read() 成员函数写 people 结构数组:

```
infile.read((char *) &people[i], sizeof(people[i]));
```

(2)应用示例

[例 10.47] 读写二进制文件。

```

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
struct person _info
{
    char name[20];
    double height;
    unsigned short age;
};
person _info people[3] = {"Qing", 1.6, 28, "Yong", 1.8, 33, "Hao", 1.82, 29};
void main(void)
{
    fstream infile;
    fstream outfile;
    outfile.open("data.dat", ios::out | ios::binary);
    if(!outfile)
    {

```

```

        cout<<"打开 data.dat 文件出错"<<endl;
        abort();
    }
    • for(int i=0;i<3;i++)
        outfile.write((char *)&people[i], sizeof(people[i]));
    outfile.close();
    infile.open("data.dat", ios::in|ios::binary);
    if(!infile)
    {
        cout<<"打开 data.dat 文件出错"<<endl;
        abort();
    }
    for(i=0;i<3;i++)
    {
        infile.read((char *)&people[i], sizeof(people[i]));
        cout <<people[i].name<<" " <<people[i].height
            <<" " <<people[i].age<<endl;
    }
    infile.close();
}

```

运行结果:

```

    Qing  1.6  28
    Yong  1.8  33
    Hao   1.82 29

```

6. 随机访问数据文件

(1) 用法

对于随机访问的文件,可以按任何顺序读记录。

①调用 `tellp()` 成员函数获取当前的流位置(该函数返回从流的始端以字节计数的当前流位置)。调用 `seekp()` 成员函数,定位到流内的一个指定的字节位置,即确定要访问的字节位置。如:

```
thefile.seekp(sizeof(data_record) * record_number);
```

其中, `data_record` 为文件使用的结构类型, `record_number` 为要定位的记录号(从 0 开始计数)。 `seekp()` 函数用于设置下一个文件读操作的具体位置。由于 `seekp()` 定位是距流(或文件)始端的绝对字节位置,所以,应用记录号乘以记录的长度,计算结果就是要访问的记录位置。由于记录号是从零开始,所以要访问的第 4 个记录就应用记录长度乘以 3,如:

```
thefile.seekp(sizeof(person_info) * 3);
```

②用 `read()` 和 `write()` 函数读写文件。

(2) 应用示例

[例 10.48] 随机访问查找文件中的任何记录。

```
#include <iostream.h>
```

```

#include <fstream.h>
#include <stdlib.h>
struct person _info
{
    char name[20];
    double height;
    unsigned short age;
};
const num _people=6;
person _info people[num _people]={ "Qing ", 1.6, 28, "Annie ", 0.7, 1,
                                     "Yong ", 1.8, 33, "Hao ", 1.82, 29};
void main(void)
{
    fstream thefile;
    person _info APerson;
    thefile.open("data.dat", ios::out|ios::in|ios::binary);
    if(!thefile)
    {
        cout<<"打开 data.dat 文件出错"<<endl;
        abort();
    }
    for(int i=0;i<num _people;i++)
        thefile.write((char *)&people[i], sizeof(people[i]));
    thefile.seekp(sizeof(person _info) * 3);
    thefile.read((char *)&APerson, sizeof(people[i]));
    cout<<APerson.name<<" "<<APerson.height<<" "
    <<APerson.age<<endl;
    thefile.seekp(sizeof(person _info) * 1);
    thefile.read((char *)&APerson, sizeof(people[i]));
    cout<< APerson.name<<" "<<APerson.height<<" "
    <<APerson.age<<endl;
    thefile.close();
}

```

运行结果:

```

Hao  1.82  29
Annie 0.7  1
age:  1

```


10.6.6 打印机的使用

1. 用法

(1) 为了将程序输出结果送到打印机上, 可以把打印设备作为一个文件打开, 大多数计算机都有与 DOS PRN 或 LPT1 设备相连的打印机, 故可用下列方式打开打印机:

```
fstream printer; //说明 fstream 类型的流标记符 printer
printer.open("PRN", ios::out); //以写的方式打开 PRN 设备
```

(2) 为使打印机前进到下一行, 应定义三个特殊的换行符:

```
const char newline[3] = {'\xA', '\xD', '\0'};
```

(3) 若把输出送到打印机时, 使用了 endl 操作器, 则必须向打印机发布一个适当的命令, 否则将得不到所期望的结果。

(4) 有的打印机接收 ASCII 值 10 作为“换行”控制命令, ASCII 值 13 作为“打印头移到左边界”控制命令, 因而 newline 的初始化设置中包含了这些代码, ASCII 值 12 被打印机解释为“换页”命令, 把常量 pageeject 初始化为 ASCII 值 12:

```
const char pageeject[2] = {'\xc', '\0'}; //12 的十六进制数为 c
```

2. 应用示例

[例 10.49] 打印机的使用。

方式 1: 打开和使用打印机。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
const char newline[3] = {'\xA', '\xD', '\0'};
const char pageeject[2] = {'\xc', '\0'};
void main(void)
{
    int age = 1;
    fstream printer;
    printer.open("PRN", ios::out);
    if(!printer)
    {
        cout<<"打印出错."<<endl;
    }
    else
    {
        printer<<"Person: Annie"<<newline;
        printer<<"Age: " <<age<<newline<<newline;
        printer<<pageeject;
        printer.close();
    }
}
```

```
}
```

运行结果:

```
Person: Annie
```

```
Age: 1
```

方式2:打印指定文件的内容。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
const char newline[3]={'\xA', '\xD', '\0'};
const char pageeject[2]={'\xC', '\0'};
void main(void)
{
    fstream printer, infile;
    printer.open("PRN", ios::out);
    infile.open("test.dat", ios::in);
    char ch;
    while(infile.get(ch))
    {
        printer<<ch;
        if(ch=='\n') printer<<newline;
    }
    printer<<pageeject;
    infile.close();
}
```

运行结果:

```
// 输出 test.dat 文件的内容
```

10.7 模板

10.7.1 模板的基本概念

1. 模板的含义

(1)模板有函数模板与类模板之分。同时支持多种数据类型参数的单独函数,称为函数模板。表达具有相同处理方法的数据对象集,称为类模板。

(2)模板在编程时并不给出相应数据的实际类型,只在编译时才由编译器使用实际的类型进行实例化,以满足编程需要,这好像在编译时,把类型作为参数传递一样,因此把这种编程方法叫做类型参数化。

(3)当编写一系列相同的重载函数时,应使用函数模板。如重载函数:

```

int max(int x, int y)
{ return(x>y)? (x):(y); }
long max(long x, long y)
{ return (x>y)? (x):(y); }
float max(float x, float y)
{ return(x>y)? (x):(y); }
double max(double x, double y)
{ return (x>y)? (x):(y); }

```

为使 max() 函数能接受 int、float、long 和 double 类型的参数,要写 4 遍 max() 函数,以让 C++ 的重载函数机制根据数据类型去确定应调用哪一个函数。

2. 模板定义的方法

(1) 对上述问题,可用函数模板来处理,只要写如下的函数模板就能替代所有的 max() 函数:

```

template <class TYPE>
TYPE max(TYPE x, TYPE y)
{ return(x>y)? (x):(y); }

```

(2) 不管作为函数或类,所有模板都是以 template 关键字和一个形参表开头。class 意为“用户定义的或固有的类型”,单词 class 与 C++ 类没有任何关系,TYPE max(TYPE x, TYPE y) 是模板。关键字 template 告诉编译器出现了一个函数模板,关键字 TYPE 是一个占位符号,它告诉编译器可用合适的数据类型替代。对于上例的模板定义,用 TYPE 代替 int,只写一个通用的 max() 函数即可,如果单目负运算符(-)对某一数据对象有定义,那么 max() 就能处理任何数据类型。

(3) TYPE 标识符只是用于标识类型参数,即参数名是可以任选的,可以选 T 作为标识符,最好选用描述性的参数名,其随后要被实际数据类型的位置标志符(如 int、double、char 或其他数据类型或指针、类结构)所替代。

(4) 一个模板说明可以列出多个参数,各个参数之间用逗号分隔,且每个参数都必须重复使用单词 class,而参数的数目无限制,例如:

```
template <class TYPE1, class TYPE2, class TYPE3>
```

(5) 创造模板的同时,也说明了准备使用的,还没有指定类型的对象,模板将使下面这个动作成为可能:即可以用一种完全通用的方法来设计函数或类,而不必预先说明所要使用的每个对象的类型。

10.7.2 函数模板的定义与使用

1. 函数模板的定义与用法

(1) 函数模板的定义

上述用重载函数计算返回最大值的例子,也可使用宏的办法来实现,如:

```
#define max(x, y) ((x)>(y)? (x):(y))
```

对于简单的函数可以这样实现,而对于较大的函数,不仅麻烦,而且易造成逻辑错误。因此,在 C++ 中已不再提倡使用宏函数,而是使用函数模板来实现,其用法是,首先设计函数模板,然

后再设计主程序来使用函数模板。

为实现上面的 max 函数的功能,我们可以使用函数模板:

```
template <class TYPE>
TYPE max(TYPE x, TYPE y)
{ return (x>y)? (x):(y); }
```

有了这个模板,就可以对一切满足模板要求的类型使用函数 max。定义函数模板后,采用“模板文件名.h”存盘,当需要使用模板时,只要在主程序开头使用#include“模板文件名.h”即可。编辑器一旦遇到使用中的模板,就会从中产生实际的模板实例,而且可以像重载普通函数那样重载函数模板。

(2)应用示例

[例 10.50] 定义使用函数模板。

```
template <class TYPE>
TYPE max(TYPE x, TYPE y)
{ return (x>y)? (x):(y); };
#include "FunctionTemplate.h"
#include <iostream.h>
void main()
{
    double dA=9.18, dB=4.21;
    cout<<"max(dA, dB)是 "<<max(dA, dB)<<endl;
    cout<<"max(9.18, 4.21)是 "<<max(9.18, 4.21)<<endl;
}
```

运行结果:

```
max(dA, dB)是 9.18
max(9.18, 4.21)是 9.18
```

[例 10.51] 重载函数模板。

```
template <class TYPE>
TYPE max(TYPE x, TYPE y)
{
    return (x>y)? (x):(y);
};
#include "FunctionTemplate.h"
#include <iostream.h>
#include <string.h>
char * max(char * a, char * b)
{
    return(strcmp(a, b)? a:b);
}
void main()
{
```

```

char * s1="Hello", * s2="Annie";
cout<<"max(s1, s2)="<<max(s1, s2)<<endl<<endl;
cout<<"max(\"Hello\", \"Annie\")是"<<max("Hello", "Annie")<<endl;
}

```

运行结果:

```

max(s1, s2)=Hello
max("Hello", "Annie")是 Hello

```

10.7.3 类模板的定义与使用

为表达具有相同处理方法的数据对象集,可以设计类模板。可以把类看作是包含某些数据类型的框架,把支持该类型的不同操作理解为从类中把数据类型分离出来。可以用单个类处理通用的数据类型 TYPE,实际上,这种类型并不是类,而只是类的描述,常称为类模板,当编译时,由编译器把类模板与某种特定数据类型联系在一起,就产生一个真实的类。

1. 类模板的定义

(1)编译器可以用类模板来定义类,一个类模板就是一个抽象的类;

(2)类模板与函数模板有些部分是相同的,如说明的方法与参数的格式等,class 在这里的含义是“任意内部类型或用户定义类型”,而 TYPE 也可能是结构或类;

(3)对函数模板与类模板来说,模板层次结构的大部分内容均相同,只是在模板说明之后,对类来说,便显示出根本性的差异,为创造类模板,在模板参数之后应有类说明。在类中可以像使用其他类型(如 int 或 double)那样使用模板参数。如:

```

template <class TYPE> //带参数 TYPE 的模板说明
class TanyTemp
{
    TYPE x, y; //类型为 TYPE 的和有数据对象
public:
    TanyTemp(TYPE x, TYPE y) // 类构造函数:x(x), y(y){}
    TYPE getx(){return x;} // 类成员函数
    TYPE gety(){return y;} // 类成员函数
};

```

其中,类模板 TanyTemp 说明了两个和有成员,即类型均为 TYPE 的数据成员 x 与 y,一旦使用类模板,它就可保存被指定类型的两个值。

(5)说明模板可以带多个参数,包括类型参数和一些其他的参数,这些参数甚至可以带缺省值,其形式为:template <参数 1, 参数 2, ..., 参数 k=缺省值 k, ..., 参数 n=缺省值 n>

(6)使用类模板的方法是在每次定义类模板的成员函数时,仍然使用关键字 template, 其形式为:

```

template<模板参数说明>
返回值类型 模板名<模板参数>::func _ name(para _ list)

```

使用模板类时,只要代进实际的参数值即可使用。

2. 类模板的对象

类模板也称为参数化类型,初始类模板时,给它传递具体的数据类型,即产生模板类。当使用模板时,编译器要自动产生处理具体数据类型的所有成员函数。如使用上述模板定义对象 iObject,并以 int 替换参数 TYPE:

```
TanyTemp <int> iObject(918,421);
```

<int>告诉编译器从模板产生一个类,并用 int 替换所有的参数 TYPE,所产生的类的名字变成 TanyTemp <int>,所定义的对象名为 iObject,两个类型值 918 和 421 传递给对象的构造函数,以初始化对象的和有数据对象。

注意:

(1)所产生类的全名为 TanyTemp<int>,这包含类括号和 int 数据类型。

(2)可以从这个模板再产生一个实例:

```
TanyTemp <double>dObject(1.24, 11.25);
```

对象 dObject 保存两个 double 值,因为又产生了一个实例 TanyTemp <double>,它是与前者毫无关系的类,相同之处只是都产生于同一个模板,只要给模板的一个实际的数据类型,就会产生新的类,且以特定类型替代模板参数。

3. 应用示例

[例 10.52] 类模板的定义与使用。

```
const int NULL=0;
template <class TYPE, int Size=10>
class Stack
{
    private:
        TYPE Buffer[Size];
        int Point;
    public:
        Stack() {Point=0;};
        ~Stack(){};
        int Empty() {return ! Point;}
        int Full() {return Point==10;}
        int Push(TYPE Object)
        {
            if(Full())
                return 0;
            else
            {
                Buffer[Point++]=Object; return 1;
            }
        }
        TYPE Pop()
        {
```

```

        return (Empty())? (NULL):(Buffer[--Point]);
    }
    void Clear() {Point=0;}
    int Count() {return Point;}
};
#include "ClassTemplate.h"
#include <iostream.h>
void main()
{
    Stack <int> stack;
    for(int i=0;i<10;++i)
        stack.Push(i);
    for(i=0;i<10;++i)
        cout<<stack.Pop()<<" ";
}

```

运行结果:

9, 8, 7, 6, 5, 4, 3, 2, 1, 0

习 题

10.1 将下面 C 程序改写为 C++ 程序。

```

#include <stdio.h>
void main()
{
    char c;
    int c1, c2;
    printf("Enter a char: ");
    scanf("%c", &c);
    c1=c-1;
    c2=c+1;
    printf("%c %d %c %d %c %d\n", c1, c1, c, c, c2, c2);
}

```

10.2 将下面 C 程序改写为 C++ 的面向对象程序。

```

#include <stdio.h>
#define PI 3.14159
void CircleArea(double Radius)
{
    double Area;
    Area=PI * Radius * Radius;
}

```

```

    printf("Area is ", Area);
}
void main()
{
    double radius;
    printf("Enter radius of a circle: ");
    scanf("%lf", &radius);
    CircleArea(radius);
}

```

10.3 写出下面程序的执行结果。

```

#include <iostream.h>
class ObjCount
{
    public:
        ObjCount(){count++;}
        int get(){return count;}
    private:
        static int count;
};
int ObjCount::count=0;
void main()
{
    ObjCount a1, a2, a3;
    cout<<a1.get()<<endl;
    cout<<a2.get()<<endl;
    cout<<a3.get()<<endl;
}

```

10.4 写出下面程序的执行结果。

```

#include <iostream.h>
class B
{
    public:
        B();
        B(int n);
};
B::B() {cout<<"B::B()"<<endl;}
B::B(int n) {cout<<"B::B("<<n<<")"<<endl;}
class D:public B
{
    public:

```

```
        D();
        D(int n);
    private:
        B b;
};
D::D() {cout<<"D::D()"<<endl;}
D::D(int n);B(n) {b=B(-n); cout<<"D::D("<<n<<")"<<endl;}
void main()
{
    D d(3);
}
```

10.5 写出下面程序的执行结果。

```
#include <iostream.h>
class B
{
    public:
        B();
        virtual void p() const;
        void q() const;
    private:
        int b;
};
B::B() {}
void B::p() const {cout<<"B::p"<<endl;}
void B::q() const {cout<<"B::q"<<endl;}
class D:public B
{
    public:
        D();
        virtual void p() const;
        void q() const;
};
D::D() {}
void D::p() const {cout<<"D::p"<<endl;}
void D::q() const {cout<<"D::q"<<endl;}
void main()
{
    B b;
    D d;
    B *pb=new B;
```

```

    B *pd=new D;
    D *pd2=new D;
    b.p(); b.q();
    d.p(); d.q();
    (*pb).p(); (*pb).q();
    (*pd).p(); (*pd).q();
    (*pd2).p(); (*pd2).q();
}

```

10.6 写出下面程序的执行结果。

```

#include <iostream.h>
class number
{
    protected:
        int val;
    public:
        number(int i){val=i;}
        virtual void show()=0;
};
class hextype:public number
{
    public:
        hextype(int i):number(i){};
        void show() { cout<<hex<<val<<dec<<endl<<endl; }
};
class dectype:public number
{
    public:
        dectype(int i):number(i){};
        void show() { cout<<dec<<val<<endl; }
};
void main()
{
    dectype d(5);
    d.show();
    hextype h(10);
    h.show();
}

```

10.7 写出下面程序的执行结果。

```

#include <iostream.h>
class A

```



```
{
    public:
        virtual void printOn()=0;
};
class B:public A
{
    public:
        virtual void printOn() {cout<<endl<<"Class B" ; }
};
class C:public B
{
    public:
        virtual void printOn() {cout<<endl<<"Class C"<<endl<<endl;}
};
void show(A *a)
{
    a->printOn();
}
void main()
{
    B *b=new B;
    C *c=new C;
    show(b);  //B::printOn()
    show(c);  //C::printOn()
}
```

- 10.8 用定义类的方式编程,输入 10 个数,把最小的数与第一个数对换,最大的数与最后的数对换。
- 10.9 用定义类的方式编程,计算一元二次方程式的实根与复根。

第 11 章

Turbo C 集成开发环境中调试程序

编写好一个程序只能说是完成程序设计任务的一半,更繁重的工作是调试程序。本章讲述 Turbo C 2.0 集成开发环境下调试 C 程序。介绍程序设计中错误的类型、集成调试器的基本概念及使用方法,并以一个实例来讲解使用集成调试器调试程序。最后给出了 C 语言程序设计中常见的错误。

建议本章授课 2 学时,上机 4 学时,自学 4 学时。

11.1 Turbo C 集成开发环境调试程序基本要领

第一章 1.3 节介绍了 Turbo C 2.0 集成开发环境下的上机基本步骤,本章讲述 Turbo C 2.0 集成开发环境下调试 C 程序。

编写程序难免会出现错误,程序调试是指对程序进行查错和排错。程序中的错误有编译错误、逻辑错误、运行错误和连接错误,而最常见的错误是编译错误和逻辑错误。利用 Turbo C 集成开发环境提供的调试程序的功能,逐步消除程序中的错误,以达到程序设计的最终目的,这一过程就是调试程序的过程。

11.1.1 纠正编译错误

1. 信息窗口(Message)

在编译和连接程序出现错误时,系统将自动激活信息窗口,并列出每个警告和错误信息,同时用高亮度光条在编辑窗口中标出程序的相应出错位置。光标位于信息窗口时,常用的快捷键有:

F5(Zoom) 将活动窗口扩展为整个屏幕,再按一次将恢复原有屏幕尺寸

F6(Switch) 切换编辑窗口、信息窗口及监视窗口为活动窗口

2. 编译错误

如果编写的 C 源程序不符合 C 语言语法规则,或程序中可能存在错误,编译时将提示错误信息,这些错误称为编译错误。编译错误包括语法错误(error)和警告错误(warning)。编译程序会在编译预处理、语法分析等阶段找出源程序中的语法错误,若出现语法错误,将不会生成目标文件(*.obj)。警告错误指出程序中有可能出现的错误,但仍会编译生成目标文件。如果程序在编译时出现错误,系统会在信息窗口显示每个错误信息,根据信息窗口提示的出错信息,可以知道错误的类型和原因,从而纠正这些编译错误。

(1)语法错误(Error)在信息窗口显示的格式是:

Error <文件名> <行>:<语法错误信息> in function <函数名>

Error: 表示本行提示的是语法错误;

<文件名>:显示当前被编译的源程序文件名。有时一个工程(项目)文件由多个源程序文件组成,这样能在多个文件中找到出现语法错误的源程序文件;

<行>:在该源程序文件中,此语法错误出现在该行;

<语法错误信息>:提示语法错误的类型及原因;

<函数名>:表示此语法错误出现在该函数内。

(2)警告错误(Warning)在信息窗口显示的格式是:

Warning <文件名> <行>:<错误信息> in function <函数名>

Warning: 表示本行提示的是警告错误;

<文件名>:显示当前被编译的源程序文件名。有时一个工程(项目)文件有多个源程序文件组成,这样能在多个文件中找到出现警告错误的源程序文件;

<行>:在该源程序文件中,此警告错误出现在该行;

<错误信息>:提示警告错误的类型及原因;

<函数名>:表示此警告错误出现在该函数内。

3. 语法错误信息

Turbo C 集成环境下,语法错误信息见(附录 G)。

4. 常见的警告错误信息

Turbo C 集成环境下,常见的警告错误信息如下:

(1)XXX declared but never used 说明了“XXX”,但是没有使用,程序中定义了变量XXX,但是没有使用该变量。编译程序遇到复合语句或函数的结束处括号时,发出此警告。

(2)XXX is assigned a value which is never used “XXX”这个变量被赋值了,但从未使用它。

(3)Code has no effect 代码无效。编译程序遇到无效的语句,如语句 $i+1;$ 。

(4)Constant out of range in comparison 比较时常量越界。如一个 int 类型变量与大于 32767 的整数相比较。

(5)Function should return a value 应使用 return 语句返回函数值。

(6)No declaration for function XXX 没有定义函数“XXX”。

(7)Non-portable pointer assignment 不允许的指针赋值。程序中将一个指针赋值给一个非指针变量,或把一个非指针值赋值给一个指针变量。

(8)Non-portable pointer comparison 不允许的指针比较。程序中将一个指针和一个非指针进行比较。

(9)Not an allowed type 不允许使用的类型。例如一个函数返回一个数组。

(10)Parameter XXX is never used 函数中说明了参数“XXX”,但从未使用它。

(11)Possible use of XXX before definition 变量“XXX”可能未赋值初值,就使用它。

(12)Possibly incorrect assignment 可能不正确的赋值。如语句 $\text{if}(i=1) i++;$ 中 $i=1$ 是赋值表达式,不是关系表达式,正确的语句可能是 $\text{if}(i==1) i++;$ 。

(13)Redefinition of XXX is not identical 源文件对宏“XXX”重新定义,新宏将替代旧宏。

(14)Structure passed by value 结构体按值传送。当结构体作为函数的参数时,为了提高程序的运行效率,通常用结构体的指针变量作为函数的参数。调用该函数时,形参可能没加地

址运算符“&”。

(15) Superfluous & with function or array 多余的运算“&”对函数或数组运算。通常地址运算符“&”对函数或是数组的运算是不必要的,应该删除该运算符。

(16) Suspicious pointer conversion 有疑问的指针转换,使指针变量指向不同的数据类型。

(17) Unreachable code 运行不到的代码。如 if (0 > 2) i++; 中表达式 i++ 永远不会被求值。

(18) Void functions may not return a value 若函数返回值类型是 void,该函数不应该返回值。

5. 纠正编译错误实例

[例 11.1] 纠正程序中的编译错误。

```
#define CODE 3456;
main()
{
    int i, j, *p;
    k = CODE/3;
    j = 1++
    k += j;
    p = i;
    k %= 3.2;
    if (k < 5)
    {
        k = 5;
        printf("%d\n", k + *p);
    }
    else
        printf("%d\n", ++ *p + k);
}
```

在 Turbo C 2.0 集成开发环境下,在编译时常用的快捷键(见附录 C)如下:

Alt+F9 仅对源程序进行编译,生成目标文件(*.obj)

F9 可对源程序进行编译并连接,生成目标文件(*.obj)和可执行文件(*.exe)

Ctrl+F9 可对源程序进行编译、连接并执行,生成目标文件和可执行文件

F5(Zoom) 将当前窗口最大化,再按一次恢复分屏状态

F6(Switch) 切换编辑窗口、信息窗口及监视窗口

[例 11.1] 经过编译后将提示下列编译错误(此处仅列出<行>和<错误信息>):

Error 5: Undefined symbol 'k'

第 5 行中未定义符号'k',变量 k 未经定义就使用。

Error 5: Expression syntax

第 5 行表达式语法错误,CODE 是一个宏(详见第六章),宏展开后多了一个分号“;”,因此错误不在本行,而是在第 1 行,应把第 1 行的分号“;”删除。

Error 6: Value required

第 6 行需要值。1++是错误的表达式,改成 k++。

Error 7: Statement missing ;

第 7 行语句缺少“;”号,错误是由于第 6 行语句未加分号“;”造成的。

Warning 8: Non-portable pointer assignment

第 8 行警告错误,不允许的指针赋值,改成 `p=&i;`。

Warning 8: Possible use of 'i' before definition

第 8 行警告错误,变量 `i` 可能未赋初值就使用,可在第 8 行前插入一行 `i=1;`。

Error 9: Illegal use of floating point

第 9 行非法浮点运算,实数不能参与取模运算,可以用强制类型转换把实型转换成整型。

Error 13: Function call missing)

第 13 行函数调用时缺少“)”号。

Error 14: Misplaced else

第 14 行 `else` 的位置不正确,这是由于 `if` 语句的复合语句缺少“}”号造成的。

Error 16: Compound statement missing }

第 16 行复合语句缺少“}”号,这还是由于 `if` 语句的复合语句缺少“}”号造成的。

这时可用光标键将信息窗口中的亮条上下移动,编辑窗口中的亮条也随着跟踪源程序中错误的位置。如果信息窗口中的错误信息太长看不见,可以左、右移动光标来水平滚动信息。按 `F5` 可以放大信息窗口,放大后编辑窗口不见了,再次按 `F5`,则恢复分屏模式。

为了改正错误,将信息窗口中的亮条置于第一个错误信息上,回车,光标移到编辑窗口中错误产生处。这样就可以在编辑窗口改正错误。如果有多处错误,可用下面两种方法之一来修改下一处错误。

第一种方法如前所述,按 `F6` 回到信息窗口,选择要修改的下一条错误信息,回车。

第二种方法不用回到信息窗口,只要按 `Alt+F8`,光标就会移至下一个错误处。按 `Alt+F7`,光标则移至前一个错误处。

按上述分析,将[例 11.1]修改成如下形式(纠正程序中连接错误见下节):

```
#define CODE 3456
main()
{
    int i, j, k, *p;
    k = CODE/3;
    j = k++;
    k += j;
    i = 1;
    p = &i;
    k %= (int)3.2;
    if (k < 5)
    {
        k = 5;
        printf("%d\n", k+*p);
    }
    else
```



```
printf("%d\n", ++ * p + k);
```

```
}
```

这样再编译、运行,程序就不会出现错误了。纠正程序中的编译错误,不一定一次将所有的错误都进行纠正,可以先纠正一条或若干条错误之后编译一下程序,若发现仍有错误再继续纠正错误。另外,编译程序提示某行出现错误,并不意味着错误一定发生在该行,错误有可能发生在前一行或前若干行上,应该综合分析,关键是读懂信息窗口提示的错误信息。

6. 编译时常见的错误

- (1) 变量没有定义就使用;
- (2) 变量未赋初值就用于表达式;
- (3) 表达式赋值给变量,但类型不相容;
- (4) 数据超界;
- (5) 除 0 错误;
- (6) 遗漏函数后面的分号“;”;
- (7) 编译预处理语句加分号“;”,如 #include, #define 等语句加了“;”号;
- (8) 定界符“'”、“”、“(”和“)”、“[”和“]”、“{”和“}”、“/ * ”和“ * /”不配对;
- (9) 将定义变量语句放在了执行语句后面,此时将提示语法错误;
- (10) 将关系符“==”误用作赋值号“=”;
- (11) 没有用 #include 指令说明文件包含,错误信息提示有关该函数所使用参数未定义;
- (12) 使用了 Turbo C 关键字作为用户标识符,此时将提示定义了太多数据类型。

11.1.2 纠正连接错误

1. 连接错误

如果编译成功,还应将目标程序和 C 的库函数连接成可执行程序(*.EXE)的文件,此时发生的错误称为连接错误。Turbo C 集成开发环境下连接程序常见错误如下:

- (1) 将 Turbo C 库函数名写错。此时,连接程序认为此函数是用户自定义函数,信息窗口显示连接错误:Linker Error:Undefined symbol <函数名> in module <程序名>;
- (2) 多个文件连接时,没有在 Project/Project name 中指定项目文件(.PRJ 文件),此时出现找不到函数的错误;
- (3) 用户自定义函数在说明和定义时类型不一致;
- (4) 由程序调用的用户自定义函数没有定义或写错函数名;
- (5) Turbo C 集成开发环境的系统配置错误,如不能打开输入文件“C0S.OBJ”。

2. 纠正连接错误实例

- (1) 不能打开输入文件“C0S.OBJ”,此时信息窗口显示连接错误:

Linker Error:Unable to open input file 'C0S.OBJ'

要解决这一问题,可以根据当前 Turbo C 2.0 系统所在的路径,正确设置菜单栏 Options/Directories 中的路径。若有多个路径,各路径之间用分号“;”隔开。

- (2) 将 Turbo C 库函数名写错,此时信息窗口显示连接错误:

Linker Error:Undefined symbol _printf in module test.c

由“Linker Error”得知是连接错误,错误信息指出:标识符“_printf”没有定义。信息窗口在

显示连接错误时没有指出错误出现在程序中哪一行,可以利用 Turbo C 提供的查找功能,找出连接错误所在位置。按 F6 由信息窗口切换到编辑窗口,光标移到文件首,按 Ctrl+QF (查找,见附录 C),系统提示“Find:”,此时键入 printf 回车,系统又提示“Option:”,回车,此时光标出现在源程序中的标识符 printf 处,将其改成 printf 即正确。注意当系统提示找什么时,不要把“_printf”前面的下划线“_”键入作为查找字符。

11.1.3 纠正逻辑错误

1. 逻辑错误

一个程序在编译和连接时都没有出现错误,执行后仍然得不到正确结果,这是由于在算法的设计过程或程序的表达式中存在错误,这种错误称逻辑错误。

纠正逻辑错误比纠正编译、连接错误要困难的多。许多逻辑错误,仅靠仔细阅读源程序,难以找到程序错误的根源。Turbo C 2.0 的集成开发环境中包含了调试程序的工具,称为集成调试器。借助于集成调试器,能够控制程序的运行,因而可以跟踪运行,查找程序的逻辑错误。如:可以在源程序的任意一行停止程序的执行,即设置断点;每次执行一条语句,在监视窗口观察程序中变量值的变化情况,即单步跟踪;甚至可以在程序运行到某一条语句时更改变量的值,以观察程序运行的中间结果有什么变化等。下面讲述有关 Turbo C 2.0 集成调试器的一些基本概念。

2. 监视窗口(Watch)

在集成开发环境中执行、调试程序时,信息窗口被监视窗口代替。可以把某些变量或表达式加入到监视窗口中,逐行执行程序、跳跃跟踪以及设置断点可以观察到监视窗口中变量及表达式值的变化情况,从而发现程序中的逻辑错误。因此,由监视窗口获取程序运行时的各种信息,是调试程序最重要的一个步骤。监视窗口中的表达式称为监视表达式。

光标位于监视窗口时,常用的快捷键有:

F5(Zoom)	将活动窗口扩展为整个屏幕,再按一次将恢复原有屏幕尺寸
F6(Switch)	切换编辑窗口、信息窗口及监视窗口为活动窗口
Insert	向监视窗口增加一个表达式
Delete	从监视窗口中删除一个表达式
Enter	编辑当前的监视表达式,修改完成后再按回车键

光标位于编辑窗口时,按 Ctrl+F7 后,将出现“Add Watch”(增加监视表达式)对话框,在此对话框中输入表达式并回车,该表达式将加入到监视窗口。有两种方式把表达式输入到“Add Watch”对话框:

(1)直接输入表达式;

(2)右移光标,光标经过的源程序表达式将逐字加入“Add Watch”对话框。

表 11.1 列出了监视窗口中表达式的默认显示格式。

表 11.1 监视窗口中表达式的默认显示格式

数据类型	显示格式
int、long、unsigned、unsinged long	用十进制整数形式显示
char、unsigned char	若 ASCII 码是(0~31)控制字符,则用转义字符显示,否则显示该字符常量
char *	显示该字符指针变量所指的字符串
其他指针类型	用地址的方式显示该指针变量的值,显示格式为: 寄存器:偏移址
float、double、long double	以小数形式或指数形式输出实型数据,若数据的宽度大于精度,自动四舍五入
数组或其他构造类型	输出每个成员的值

监视窗口中表达式值的显示格式可以按系统默认的方式显示,也可以利用“格式说明符”设置其显示格式。设置监视窗口表达式值的显示格式,应使用如下的形式:

表达式,[格式说明符 1][格式说明符 2][...]

表 11.2 列出了监视窗口中表达式的格式说明符及其显示格式。

表 11.2 监视窗口中表达式的格式说明符及其显示格式

格式说明符	显示格式
c	显示 ASCII 码值为 0~31 的控制字符,而不是显示转义字符。对字符及字符串有效。
d	所有整数值以十进制数显示,对整数表达式及含有整数的数组、结构体有效。
f[n]	n 是 2~18 之间的整数,显示 n 位有效位的实数,对实数有效。
h 或 x	所有整数值以十六进制数显示,显示的前缀为 0x,对整数表达式及包含整数的数组、结构体均有效。
m	监视表达式必须是变量或能被赋值的表达式,从该变量的内存地址开始显示内存的内容。系统默认每个字节以两位十六进制数显示。若与 d 格式说明符联用,则每个字节以十进制数显示;若与 h 或 x 格式说明符联用,则每个字节以两位十六进制数显示;若与 c 或 s 格式说明符联用,将以字符串的形式进行显示。
p	以段地址:偏移址格式显示指针,仅对 far 及 huge 指针有效。
r	显示每个结构体的名字和值。对结构体有效。
s	控制字符(ASCII 码值为 0~31)用转义字符来显示,由于这是系统默认的字符及字符串的显示格式,所以只有和 m 格式说明符联用才有效。
正整数	指定数组元素或结构体变量连续显示的个数。对数组及结构体有效。

表 11.3 举例说明监视窗口中表达式值的显示格式。为了说明监视窗口的格式说明符的用法,假设定义了如下的结构体:

```
struct atype
{ char *p;
  int b[3];
  char ch;
};
```


表 11.3 举例说明监视窗口中表达式值的显示格式(struct atype 定义如上)

定义变量	监视表达式	监视表达式的值 (整数用十进制)	监视窗口显示 值的格式	说 明
int i;	i	123	123	系统默认方式
int i;	i, x	123	0x7B	显示十六进制数
float f;	f	123.456789	123.4568	系统默认方式
float f;	f, f5	123.456789	123.46	以精度 5 位显示
double db	db	12345.678966	12345.678966	系统默认方式
double db	db, f4	12345.67896	1.235e+4	以精度 4 位显示
char ch;	ch	'A'	'A'	系统默认方式
char ch;	ch	'\36'(八进制)	'\x1E'	系统默认方式
char ch;	ch, c	'\36'	▲	显示控制符
int i, *q=&i;	q	FF88	DS:FF88	系统默认方式
int i, far *q=&i;	q, p	5E88;FF86	5E88(DS);FF86	p 说明符显示指针
char s[10];	s	"abcde"	"abcde"	系统默认方式
int a[5];	a	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}	系统默认方式
int a[5];	a[1], 3	{1, 2, 3, 4, 5}	2, 3, 4	显示 a[1] 开始之后的 3 个元素值
int a[5];	a[1], 5m	{1, 2, 3, 4, 5}	02 00 03 00 04	显示 a[1] 开始之后的 5 个内存单元的数据
struct atype su;	su	{"zxy", {9, 8, 7}, 'B'}	{"zxy", {9, 8, 7}, 'B'}	系统默认方式
struct atype su;	su, r	{"zxy", {9, 8, 7}, 'B'}	{ p: "zxy", b: {9, 8, 7}, ch: 'B' }	显示结构体成员的 值, 同时显示成员名

应当注意, 当调试进入函数时, 例如有下面的函数定义:

```
int sort( int a[ ], int n);
{
    ...
}
```

监视窗口中观察数组名 a, 显示的是该实参数组的首地址, 不会显示各元素的值, 这时可用 "a[0], 5" 作监视表达式, 就会显示由 a[0] 开始连续 5 个元素的值。这一方法同样适用于结构体数组或结构体指针作形参的情况。

3. 单步调试

调试程序时, 按上述方法把要观察的变量和表达式加入监视窗口, 每按一次 F7 键, 就执行一行程序, 这称为单步调试程序。只要不断地按 F7 键, 就可以从 main 函数的第一行开始不断地向下执行每行程序, 执行到哪一行, 就把该行高亮度显示。在单步执行的过程中, 观察监视窗口中的变量和表达式的变化情况, 以便找出程序中的逻辑错误。在调试过程中可以编辑、增加以及删除监视窗口中的表达式。按 F6 切换编辑窗口、监视窗口。使用 F7 键还可以跟踪到用

户定义的函数中。

4. 跳跃跟踪

(1) 如果在使用单步跟踪时, 不想进入一个已调试好的函数, 可以按 F8 键, 集成调试器就把函数当做一行来处理, 就不会对该函数进行调试。

(2) 如果想跳过某些已经确定无错误程序段后, 再执行单步跟踪, 将光标移动到要单步跟踪的行上, 按 F4 键程序将直接执行到该行停下, 并将该行高亮度显示, 然后再按 F7 进行单步跟踪。

(3) 按 Ctrl+F8 设置断点行, 该行将用红底白字显示, 程序执行到每个断点行都将暂停下来。然后再进行以上各种调试(F7, F8, F4)。注意, 断点至少应含有一条可执行语句, 在 Turbo C 2.0 集成开发环境中, 最多可设置 21 个断点行。

5. 运算窗口

按 Ctrl+F4 可以调用运算窗口。Turbo C 2.0 的运算窗口如图 11.1 所示。

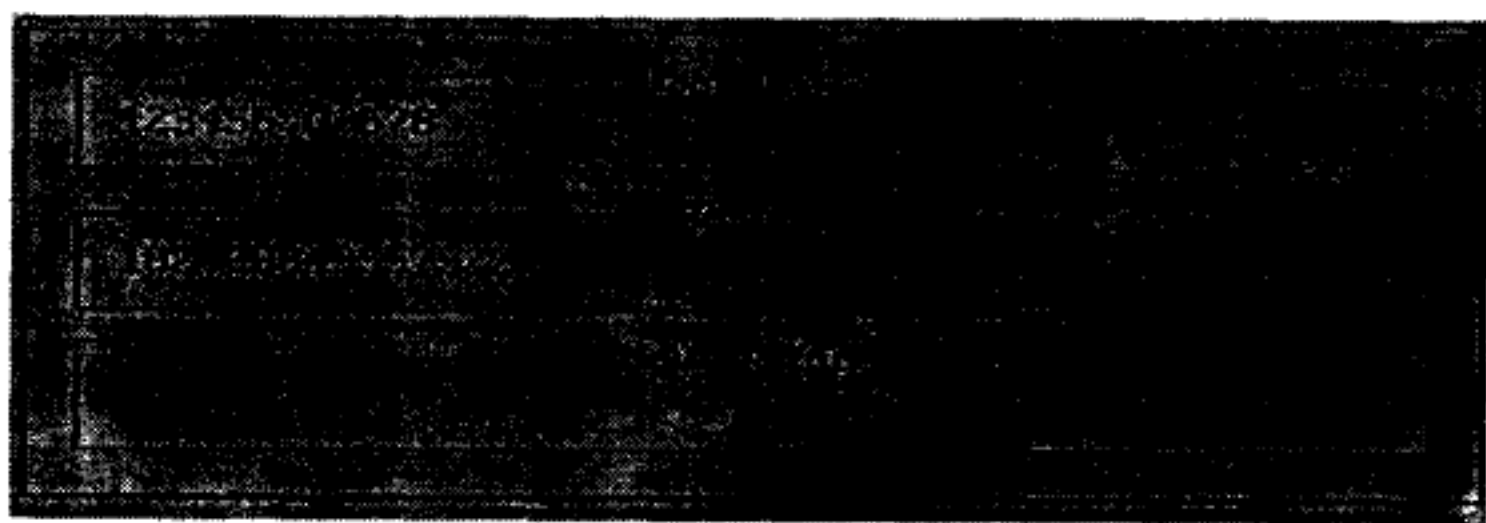


图 11.1 Turbo C 2.0 的运算窗口

利用运算窗口可以进行表达式计算以及改变变量当前的值。

(1) 计算表达式的值

可以向监视窗口添加表达式, 并计算出该表达式的值。有时需要临时查看某些变量的值或在调试程序时临时计算某表达式值, 用监视窗口添加表达式后又要删除, 比较麻烦, 这时可调用运算窗口。运算窗口由三个部分组成, 即 Evaluate(计算)、Result(结果)和 New value(新值)。在 Evaluate 对话框中键入表达式并回车, 这时在 Result 对话框中将显示该表达式的值。

(2) 修改变量的值

运算窗口最强大的功能是在程序运行期间对变量的值加以修改, 这在调试程序时是非常有用的, 比如可以改变某循环变量的值, 使程序不必逐步执行就可满足某些特定的条件。在 Evaluate 对话框中键入某变量名, 在 New value 对话框中键入新值并回车, 则该变量的值被改变成新输入的值。

6. 用户屏幕

调试时, 如果程序向屏幕输出某些结果, 只能看见屏幕闪了一下, 看不清屏幕显示的内容, 这时可以按 Alt+F5 组合键从 Turbo C 2.0 的主屏幕切换到用户屏幕, 就可看见 DOS 环境下的程序输出结果, 再按任意键返回 Turbo C 2.0 的主屏幕。

7. 程序复位

调试程序时, 发现错误并改正后, 通常应重新运行程序。在 Turbo C 集成开发环境的主屏幕中按 Ctrl+F2 可以使程序复位, 即中止当前的程序运行和调试, 释放已分配的空间并关闭文件, 下次运行将重新开始执行程序。

8. 中断程序运行

调试程序时,由于程序中存在某些错误(如死循环),程序运行后一直停留在用户屏幕,无法回到 Turbo C 集成开发环境的主屏幕中,这时可按 Ctrl+Break 组合键,中断程序运行,再按 Esc 键,光标就回到 Turbo C 集成开发环境的主屏幕中。

11.2 调试程序实例

1. 调试前的准备

在首次使用 Turbo C 集成调试器进行程序调试时,应该先检查一下 Turbo C 集成开发环境的配置是否与下述设置相同,如果不相符,应注意改变过来。

(1)在菜单栏“Options/Directories/Include directories”中检查头文件目录(一般为“盘符:\tc\include”)设置是否正确;

在菜单栏“Options/Directories/Library directories”中检查库文件目录(一般为“盘符:\tc\lib”)设置是否正确。

(2)把菜单栏的“Debug/Source debugging”和“Options/compiler /Code generation/OBJ debug information”设置为 ON。

(3)在菜单栏“File/Change dir”中,把将要调试的源程序所在目录设置为当前目录。

2. 调试程序实例

[例 11.2] 由键盘输入两个字符串,输出这两个字符串的最长公共子串。例如:输入以下两个字符串:s1 为“5 aabcdaklsa”,s2 为“xaaabcdalkls6” 则两个字符串的最长公共子串是 aabcd a。

编写一函数 void fun(char * s1, char * s2, char * strsame),将字符串 s1 与字符串 s2 的最长公共子字符串存入 strsame 所指的字符数组中。本题算法分析如下:

(1)作为外层循环,指针变量 start1 扫描字符串 s1,每循环一次,start1 指向下一个字符;

(2)作为第二层循环,指针变量 start2 扫描字符串 s2,每循环一次,start2 指向下一个字符;

(3)内层循环,用指针变量 p1 从 start1 开始,同时指针变量 p2 从 start2 开始,向后逐一比较 * p1 与 * p2 两字符,直到找到一个不同的字符为止,即找到了一个公共子串,长度为 len。若此公共子串比前面找到的公共子串长,则记录该公共子串的位置 maxstart = start2 和长度 maxlen = len,如图 11.2 所示;

(4)重复步骤(1)、(2)、(3)直至退出所有循环语句;

(5)输出由 maxstart 所指的字符开始之后连续 maxlen 个字符到 strsame 所指的字符数组中。



图 11.2 查找两个字符串中的公共子串

为了供调试使用,程序中包含了几处错误。在每行程序前加行号以方便阅读,程序如下:

```
/* 1行 */ #include "stdio. h"
```

```

/* 2行 */   main()
/* 3行 */   {
/* 4行 */       void fun(char * s1, char * s2, char * strsame);
/* 5行 */       char str1[300]="", str2[300]="", substr[300]="";
/* 6行 */       printf("请输入两个字符串:");
/* 7行 */       scanf("%s%s", str1, str2);
/* 8行 */       fun(str1, str2, substr);
/* 9行 */       printf("两个字符串中的最长公共子串是:")
/* 10行 */      puts(substr);
/* 11行 */      getch();
/* 12行 */  }
/* 13行 */  void fun(char * s1, char * s2, char * strsame)
/* 14行 */  {
/* 15行 */      char * p1, * p2, * start1, * start2, * maxStart;
/* 16行 */      int len, maxlen=0;
/* 17行 */      start1=s1;
/* 18行 */      start2=s2;
/* 19行 */      while(* start1)    /* 用指针变量 start1扫描字符串 s1 */
/* 20行 */      {
/* 21行 */          while(* start2) /* 用指针变量 start2扫描字符串 s2 */
/* 22行 */          {
/* 23行 */              p1=start1; /* 用指针变量 p1从 start1开始扫描字符串
/*                               s1, 找出公共子串 */
/* 24行 */              p2=start2; /* 用指针变量 p2从 start2开始扫描字符串
/*                               s2, 找出公共子串 */
/* 25行 */              len=0;    /* len 记录公共子串的长度 */
/* 26行 */              while(* p1++ == * p2++) len++; /* 求出公共子
/*                                               串的长度 */
/* 27行 */              if (len>maxlen) /* 记录最长公共子串 */
/* 28行 */              {
/* 29行 */                  maxlen=len;
/* 30行 */                  maxstart=start2;
/* 31行 */              }
/* 32行 */              start2++;
/* 33行 */          }
/* 34行 */          start1++;
/* 35行 */      }
/* 36行 */      /* 以下输出最长公共子串到 strsame 所指的字符数组中 */
/* 37行 */      for(p1=strsame, p2=maxstart; p2<maxstart+maxlen;
p1++, p2++);

```

```

/* 38行 */      *p1 = *p2;
/* 39行 */      *p1 = '\0';
/* 40行 */      }

```

(1) 改正编译错误和连接错误

使用 Turbo C 集成环境调试程序,通常应先改正编译错误和连接错误。首先将[例11.2]程序输入编辑窗口。按 Ctrl+F9 运行该程序,这时信息窗口将提示如下信息:

Error...7: Function call missing)	第7行调用函数时,括号“)”遗漏了
Error...10: Statement missing ;	第10行(实际第9行)语句的分号“;”遗漏了
Warning...26: Possibly incorrect assignment	警告:第26行有可能是不正确的赋值
Error...30: Undefined symbol 'maxstart'	第30行未定义标识符“maxstart”
...	

改正上述三处错误:回车,光标回到编辑窗口,在第7行函数调用语句分号前加一个括号“)””;在第9行语句未加一个分号“;”;在第15行将变量名 maxStart 改成 maxstart。对于26行的警告错误暂时不作处理。

再次按 Ctrl+F9 运行该程序,信息窗口又提示如下信息:

Linker Error: Undefined symbol '_printf'... 连接错误:未定义标识符'_printf'

按 F6 切换到编辑窗口,光标移到文件头,用查找功能键 Ctrl+QF 找到 printf,发现第9行的格式输出语句书写错误,将“printf”改正成“printf”。

(2) 改正逻辑错误

调试程序的关键在于找出程序中的逻辑错误,并加以纠正。纠正逻辑错误比纠正编译错误困难得多。比再次按 Ctrl+F9 运行该程序,运行时输入、输出结果如下:

请输入两个字符串: 5aabcdaklsa xaaabcdalkls6

两个字符串中的最长公共子串是:

上述运行结果未找到公共子串,显然与题意不符,说明程序中存在逻辑错误。按 Ctrl+F7 加入5个监视表达式到监视窗口:

len、start1、start2、p1、p2(每个监视表达式占一行)

光标移动到第19行 while(*start1) 处,按 F4 运行到光标处。显示用户屏幕,并作如下输入:

请输入两个字符串: 5aabcdaklsa xaaabcdalkls6

程序运行回到 Turbo C 集成开发环境,第19行程序高亮度显示,表示当前程序运行到了第19行。观察监视窗口 start1 与 start2 指向的字符串如下:

start1: "5aabcdaklsa"

start2: "xaaabcdalkls6"

说明输入过程没有错误。在第26行 while(*p1++ == *p2++) len++; 处按 Ctrl+F8 设置1个断点行。按 Ctrl+F9 程序运行到第26行暂停,观察监视窗口 start1、start2、p1、p2 指向的字符串如下:

start1: "5aabcdaklsa"

start2: "xaaabcdalkls6"

p1: "5aabcdaklsa"

p2: "xaaabcdalkls6"

说明 p1开始时指向 start1, p2开始时指向 start2, 按 F7单步运行程序运行, 程序运行到第27行, 观察监视窗口 len 的值及 p1、p2指向的字符串如下:

```
len:12
```

```
p1:""
```

```
p2:""
```

p1、p2均指向字符串结束标志。第26行程序本应找出两个字符串的公共子串, 并用变量 len 计数公共子串的长度, 这时 len=12显然是错误的。把第26行的 while 语句分写成如下2行:

```
while (*p1++ == *p2++)
```

```
len++;
```

再把断点行移至语句 len++, 按 Ctrl+F2使程序复位, 再次按 Ctrl+F9运行程序, 然后按 F7单步运行, 观察监视窗口发现 start2所指的字符串不断地复制到 start1所指的字符串位置上, 这是由于把关系运算符“==”写成“=”所造成的; 另外在单步运行时, 还发现当 p1或 p2指向字符串结束标志时, 就不应该再比较是否是公共子串了, 经过上述分析把原来程序的第26行改写成如下形式:

```
while (*p1 && *p2 && p1++ == p2++) len++;
```

(3) 继续调试

当程序的一处或多处逻辑错误被纠正之后, 按 Ctrl+F2使程序复位(在运行前可按 F2先保存文件), 再次按 Ctrl+F9运行程序, 这次还是未找到公共子串。光标移到第34行 start1++; 处按 F4程序运行到光标所在行暂停, 然后按 F7单步运行, 观察监视窗口 start2总是指向空的字符串, 这是由于 start2只在第一次外层循环时赋初值, 由题义知, 每当 start1指向 s1的下一个字符时, start2要从头到尾扫描一遍 s2中的字符串。因此第18行的赋值语句 start2 = s2; 应搬移到第21行 while(*start2)的上面一行。

按 Ctrl+F2使程序复位, 再次按 Ctrl+F9运行程序, 这次仍然未输出公共子串。让程序运行到37行暂停, 按 Ctrl+F4分别计算表达式: maxlen 与 maxstart(当然也可把它们加入到监视窗口中), 发现两个字符串的最长公共子串找到了: “aabca”(因为 maxlen=6)。这时可以判定问题出在没有把正确的结果输出到 strsame 所指的字符数组中, 继续调试程序发现第37行的 for 语句之后多了一个分号“;”, 将其删除。

按 Ctrl+F2使程序复位, 再次按 Ctrl+F9运行程序, 这次正确的结果终于出现了。可以再多试几组数据, 测试一些特殊的情况, 以检验程序是否有其他逻辑错误或运行错误。改正后的 [例11.2] 程序如下:

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
void fun(char *s1, char *s2, char *strsame);
```

```
char str1[300]="", str2[300]="", substr[300]="";
```

```
printf("请输入两个字符串:");
```

```
scanf("%s%s", str1, str2);
```

```
fun(str1, str2, substr);
```

```
printf("两个字符串中的最长公共子串是:");
```

```
puts(substr);
```



```

    getch();
}
void fun(char *s1, char *s2, char *strsame)
{
    char *p1, *p2, *start1, *start2, *maxstart;
    int len, maxlen=0;
    start1=s1;
    while(*start1) /* 用指针变量 start1 扫描字符串 s1 */
    {
        start2=s2;
        while(*start2) /* 用指针变量 start2 扫描字符串 s2 */
        {
            p1=start1; /* 用指针变量 p1 从 start1 开始扫描字符串 s1,
                        找出公共子串 */
            p2=start2; /* 用指针变量 p2 从 start2 开始扫描字符串 s2,
                        找出公共子串 */
            len=0; /* len 记录公共子串的长度 */
            while(*p1 && *p2 && *p1++ == *p2++)len++;
            /* 求出公共子串的长度 */
            if (len>maxlen) /* 记录最长公共子串 */
            {
                maxlen=len;
                maxstart=start2;
            }
            start2++;
        }
        start1++;
    }
    /* 以下输出最长公共子串到 strsame 所指的字符数组中 */
    for(p1=strsame, p2=maxstart; p2<maxstart+maxlen; p1++, p2++)
        *p1 = *p2;
    *p1 = '\0';
}

```

找出程序中的错误并加以改正是程序设计的一个重要步骤,对 C 语言的初学者来说,这并不容易。初学者在上机实践中,要尽可能使用 Turbo C 的集成调试器来纠正程序中的错误,只有在实践中不断积累调试程序的经验,才能真正熟练掌握用 C 语言编程。

在编写程序时,注意以下几个方面,可以减少程序中的错误,也使查找错误容易许多:

- (1) 书写每个复合语句都应缩进,变量名尽量见名知义,程序中多加注释。
- (2) 把一个复杂语句分写成多个简单语句,这样调试、阅读、修改及查错都容易。
- (3) 编写函数的功能应尽量单一。一个函数中的源程序代码不应过长,一般在 30 行以内。超

过30行的函数,最好按函数的功能,把其中的代码抽出来,再编写成另一个函数,并调用它,形成多级调用的方式。这样可以逐个调试函数,每个函数的代码都不长,减少了调试程序的难度。在有些应用中,函数的结构简单且整体化强,也不必强求函数一定要在30行以内,完全可以多于30行。

(4)函数中尽量少用全程变量,应尽量由形参向函数传入数据。

(5)每个变量在使用前都应考虑到当前的初值是什么,应特别注意对指针变量赋初值。

11.3 调试程序命令和热键小结

Turbo C 2.0集成环境下,许多调试程序命令和其他菜单命令都可以使用热键或组合键来完成。表11.4给出了 Turbo C 2.0集成调试器的调试命令和热键。

表11.4 Turbo C 2.0集成调试器的调试命令和热键

热键	菜单命令	使用说明
Ctrl+F7	Break/Watch/Add watch	光标在编辑窗口,增加一个监视表达式。或光标在监视窗口按 Insert 键。
	Break/Watch/Delete watch	删除一个监视表达式。或光标在监视窗口按 Delete 键。
	Break/Watch/Edit watch	编辑监视表达式。或光标在监视窗口按回车键。
	Break/Watch/Remove all watch	删除所有的监视表达式。
Alt+F7		在编辑窗口中,光标移至上一个编译或连接错误处。
Alt+F8		在编辑窗口中,光标移至下一个编译或连接错误处。
F7	Run/Trace into	逐行执行当前行语句,并可以跟踪进入到函数中。
F8	Run/Step over	逐行执行当前行语句,但不能跟踪进入到调用的函数。
F4	Run/Go to cursor	执行程序到光标所在行。
Ctrl+F8	Break/Watch/Toggle breakpoint	在光标所在行设置或清除断点行。
	Break/Watch/Clear all breakpoint	清除程序中的所有的断点。
	Break/Watch/View next breakpoint	显示下一个断点。
Ctrl+F2	Run/Program reset	中止程序运行和调试,释放已分配的空间并关闭文件。
Ctrl+F4	Debug/Evaluate	计算表达式,允许修改变量的值。
	Debug/Find function	在编辑窗口中查找函数定义。
Ctrl+F3	Debug/Call stack	显示调用栈,可显示函数调用和被调用的层次。
Alt+F5	Run/User screen	显示用户屏幕,按任意键返回到集成环境主屏幕。
F5		将当前窗口在全屏幕和分屏幕之间切换,按任意键返回。
F6		在编辑窗口和监视窗口或信息窗口之间切换。
Alt+F6		如果光标在编辑窗口,就把上一个文件调入编辑窗口;如果下部窗口是活动的,就在监视窗口和信息窗口之间进行切换。
Ctrl+F9	Run/Run	运行程序,必要时编译、连接源文件。
Alt+F9	Compli/Compil to OBJ	编译源程序生成目标程序。
F9	Compli/Make EXE file	编译生成目标文件后再连接生成可执行文件。
	Project/Remove messages	删除信息窗口中的内容。
Ctrl+Break		中断程序运行,从用户屏幕回到 Turbo C 集成开发环境主屏幕。

11.4 Turbo C 程序的常见错误

C 语言功能强大,C 语言程序简洁、紧凑;C 语言使用方便、灵活,可直接对硬件进行编程,是当今最受欢迎的程序设计语言之一。同时,用 C 语言编写出的程序运行效率较高、占用内存少。利用标准库函数和用户自定义函数,可以使程序设计模块化。

另一方面,正是由于 C 语言的灵活性,使 C 语言的编译程序对语法的检查不如其他的高级语言严格,它要求程序设计人员设法保证程序的正确性,这给初学者掌握 C 语言带来了一定的困难。初学者要不断积累经验,不断地提高程序设计和程序调试的水平。

下面分八个方面介绍用 C 语言编程可能出现的错误。

11.4.1 使用变量容易出现的错误

1. 忘记定义变量

C 语言要求变量都必须先定义、后使用。

2. 标识符的大小写字母混用

C 语言标识符的大小写是有区别的。C 语言程序中把 Area 和 area 当作不同变量。

3. 变量或指针变量没对其赋初值,就用于表达式

例如:

```
int a[10],b;  
int *p, i=1;  
b=a[8];  
*p=i
```

以上程序段在编译时会出现警告信息,指针变量 p 与数组元素 a[8]均未赋初值。

11.4.2 编写表达式容易出现的错误

1. 将赋值号“=”误用作关系符“==”

BASIC 语言中,把“=”既作为赋值运算符,又把它作为关系运算符的“等于”。C 语言中,“=”是赋值运算符,“==”是关系运算符。例如,经常会出现下面的错误用法:

```
if( a=b )...
```

在 C 语言中,将(a=b)当做赋值表达式,它将 b 的值赋值给 a,然后判断赋值表达式 a=b 的值是否为 0,如果值是非零,则执行 if 语句的语句体;否则,将执行 if 语句的后继语句。

2. 使用“++”或“--”运算符时易犯的错误

例如下面的程序段:

```
int a[5]={1,2,3,4,5}, *p;  
p=a;  
printf("%d\n", *p++);
```

上述程序中,运算符“*”和“++”运算优先级相同,结合方向是“从右到左”,表达式

p++等价于(p++)，先输出p指针所指向的内容(即a[0]的值1)，然后使指针p指向a[1]。如果是*(++p)，则指针p先指向数组a[1]，然后输出元素a[1]的值。如果要使指针变量p所指的内容加1(即a[0]++)，则必须用表达式(*p)++，此处圆括号不能省略，这是初学者最易犯的错误。

3. a>>3操作并不能改变a的值

例如有下面的程序段：

```
unsigned a, b;
a=19;
b=a>>3;
printf("%u, %u\n", a, b);
```

输出结果是:19, 2

表达式a>>3不会使操作对象a的值发生变化。在作a=a>>3;操作时，变量a的值才发生变化。其他的位运算符也有类似的情况，位运算不会改变操作对象的值。

4. 错把位与运算符“&”当做逻辑与运算符“&&”

例如:if (score>=80 & score<90) printf("良\n");

此处将位与运算符“&”当做了逻辑运算符“&&”。

5. 错把整数除做为实数除

如下程序段输出变量a, b的平均值：

```
int a=10, b=15;
float f;
f=(a+b)/2;
printf("%f\n", f);
```

输出结果是:12.000000

正确的结果是12.500000。错误在第三条语句f=(a+b)/2;是整数除，应该改为实数除：f=(a+b)/2.0;、f=1.0*(a+b)/2;或者f=(float)(a+b)/2;。

6. 定界符不配对造成错误

C语言要求“”、“”、“(”和“)”、“[”和“]”、“{”和“}”、“/*”和“*/”配对使用。例如下面的语句是由于定界符不配对引起的错误：

```
ch='A; /* 缺少了一个单引号"'" * /
printf("Good morning !\n); /* 缺少了一个双引号"\"" * /
if ( (fp=fopen("c:\\windows\\ab.txt")==NULL){...} /* 缺少了一个")" * /
```

7. 表达式赋值给变量，但类型不相容

例如：

```
int *p, a;
p=1234;
a=p;
```

指针变量不能用整数赋值，也不能把一个指针赋值给一个整型变量。

8. 除0错误

程序设计时，要考虑到除0错误。例如以下程序可以防止除0错误发生：

```
if ( b!=0 ) c = a/b;
```

```
else printf("出错信息...\n");
```

9. 数据超界

数值超过了该数据类型可表示的范围。例如：

```
int i;  
char ch;  
i=32768 ;  
ch=129;  
printf("%d,%d\n", i, ch);
```

以上程序段运行结果是：-32768，-127。

在 IBM PC 兼容微型计算机上编译 C 程序，一个整型数据(int)规定为两个字节，其表示范围从-32768到32767；一个字符型数据(char)规定为1个字节，其表示范围从-128到127，所以变量 i 与变量 ch 所赋的值超过了表示范围。上述程序段应改成：

```
long int i;  
unsigned char ch;  
i=32768 ;  
ch=129;  
printf("%ld,%d\n", i, ch); /* 注意这里必须用%ld 格式说明符来输出长整型变量 */
```

此时运行结果是：32768，129。

11.4.3 使用语句容易出现的错误

1. 遗漏语句后面的分号“;”

C 语言规定，语句必须以分号结束，分号是 C 语句不可缺少的一部分，初学者往往忽略了这个分号。例如：

```
i=123  
j=456;
```

又如在复合语句中漏写最后一个语句的分号：

```
{  
    a += b;  
    b ++  
}
```

2. 语句后不该加分号的地方加了分号

例如：

```
for (i=0; i<10; i++);  
    printf("%d ", a[i]);
```

又如：

```
if ( a<b );  
    min = a;
```

再如：

```

fun(int a[], int n);
{
    int i;
    :
}

```

学过 Pascal 语言的读者很容易犯这样的错误。

3. 编译预处理语句加分号“;”

如 #include、#define 等语句后不该加分号却加了“;”分号。

4. 将定义变量语句放在了执行语句后面

例如：

```

int i;
i=0;
float s=1.0;

```

此时将提示语法错误。

5. 复合语句忘记加花括号

例如下面程序段求 $s=1+2+3+\cdots+100$ ：

```

i=1; s=0;
while (i<=100)
    s += i;
    i++;

```

上面的 while 循环语句将进行死循环，因为语句 $i++$ ；不在循环体内。

6. switch 语句中没有使用 break 语句

在 switch 语句的各分支中忘记使用 break 语句跳出多分支语句。例如：

```

switch(grade)
{ case 'A' : printf("优\n");
  case 'B' : printf("良\n");
  case 'C' : printf("及格\n");
  case 'D' : printf("不及格\n");
  default ; printf("\n");
}

```

前4个 case 语句的最后都应该加一条 break; 语句。

7. 多层条件语句的 if 和 else 不配对

例如下面程序段，输出变量 a、b、c 中的较大者：

```

int a=5, b=2, c=3;
if (a>b && a>c ) printf("%d\n", a);
    if(c>a && c>b) printf("%d\n", c);
    else printf("%d\n", b);

```

上面程序将输出：5 2。这是由于第2个 if 语句前缺少 else 造成的。上面程序段应改成：

```

int a=5, b=2, c=3;
if (a>b && a>c) printf("%d\n", a);

```



```

else if(c>a && c>b) printf("%d\n", c);
else printf("%d\n", b);

```

8. 混淆 break 语句和 continue 语句的作用

例如以下程序段,要输出500以内所有能被7整除的数:

```

for(i=7; i<=500; i++)
{
    if ( i%7 !=0 ) break;
    printf("%d", i);
}

```

上述程序中 break 语句应该改成 continue 语句。

11.4.4 使用数组容易出现的错误

1. 误把数组说明时所定义的元素个数作为最大下标值使用

C 语言规定使用数组元素时,下标从0开始,即下标值的下界为0,而下标的上界值是数组定义时元素个数减1。例如,若定义 `int a[10];`,则使用 `a[10]` 元素是错误的。

2. 定义或使用数组的方式不对

数组定义或使用数组元素时必须要用方括号,对二维数组或多维数组的每一维下标变量都必须分别用方括号括起来。例如以下写法都是错误的:

```
int a(10),b[3,5];
```

3. 混淆数组名与指针变量

在 C 语言中,数组名是数组的首地址,它的值是一个常量,不能被修改。例如,在以下程序段中,用 `a++` 是不合法的。

```

int i, a[10];
for (i=0; i<10; i++)
    scanf("%d", a++);

```

4. 向数组名(地址常量)赋值

例如:

```

char s[100];
s="Hello! ";

```

上面的错误是将数组和指针的概念混淆造成的。数组名是地址常量,数组在定义的同时可以对其初始化,例如:`char s[100]="Hello!"`;,但不能对数组名赋值。而指针可以在其定义之后再对其赋值,这时是将字符串的首地址赋给指针变量。

5. 数组初始化越界

例如:

```
char s[6]="不及格";
```

此处错误在于定义数组时所开辟的空间不足,给数组 `s` 所开辟的存储空间只有六个字节,而对其初始化操作有七个字符(字符串结束标志 `'\0'` 还要占一个字节)。可以不指定数组的元素个数,让系统自动确定数组的大小:`char s[]="不及格"`; 相当于 `char s[7]="不及格"`;

6. 误把数组名作为全体数组元素

C 语言中,不能对数组进行整体赋值。例如下面程序段想将 a 数组的元素赋值给 b 数组对应的元素:

```
int a[5]={5, 6, 7, 8, 9}, b[5];
b=a;
```

上述赋值是错误的, b 是数组的首地址,是常量,不能被赋值。正确的程序段是:

```
int i, a[5]={5, 6, 7, 8, 9}, b[5];
for (i=0; i<5; i++) b[i]= a[i];
```

11.4.5 使用库函数容易出现的错误

1. 调用 scanf 函数时,变量名前漏写地址运算符 &

例如:

```
float x, y;
scanf("%f%f", x, y);
```

2. 调用 scanf 函数时,规定输入实型数据的小数位

例如:

```
float x;
scanf("%6.2f", &x);
```

使用格式输入函数 scanf 时,不能规定输入实型数据的小数位,因此执行上述 scanf 语句时出错,将不会对变量 x 输入数据,直接执行 scanf 的之后的语句。但是,调用格式输入函数 scanf 时,可以规定输入实型数据的长度(小数点、正负符号均计入长度)。例如以下调用是正确的:

```
scanf("%6f", &x);
```

3. 调用 scanf 函数时,输入的数据格式与格式化字符串的要求不一致

调用 scanf 函数输入数据时,必须注意键盘输入的数据格式要与 scanf 语句中的格式化字符串一一对应匹配。如:

```
scanf("%d,%d", &a, &b);
```

若从键盘输入数据: 5 6 回车,是不正确的输入方式。数据5和6之间应当加逗号。

4. 调用 printf 函数时,输出数据的类型与格式说明符不匹配

例如有以下说明语句:

```
long int a=56789;
int b=123;
printf("%d, %d\n", a, b);
```

上述程序段运行结果是:-8747, 0,这显然是错误的输出结果。printf 语句中输出长整型变量,应该使用格式说明符"%ld"。

5. 格式化输入输出时,变量的类型与格式说明符规定的类型不一致

例如:

```
double d;
scanf("%f", &d);
```

上面程序段中,d 为双精度实型变量,正确调用方式为:scanf("%lf",&d);。

6. 动态分配内存后未调用 free 函数释放内存空间

调用动态内存分配函数 malloc()或 calloc()分配的内存空间使用之后,如果没有调用 free()函数释放所分配的内存空间,会导致多次动态分配内存之后发生死机现象,其原因是所分配的空间占用了操作系统代码所在的内存空间,破坏了操作系统。

7. 使用 abs 函数与 fabs 函数时混淆

abs(x)函数与 fabs(x)函数均返回 x 的绝对值。abs(x)函数用于 x 是整型表达式的情况,其返回值也是整型;fabs(x)函数用于 x 是实型表达式的情况,其返回值是实型。

8. 调用库函数,但程序中没有使用相应的库文件包含

例如下程序:

```
main()
{
    printf("%f\n", sqrt(25)); /* 求25的平方根 */
}
```

上述程序编译、连接不会提示错误信息,但是运行后将输出错误结果。这是因为使用 sqrt 函数必须使用库文件包含 #include "math.h"。

11.4.6 使用自定义函数容易出现的错误

1. 误将函数形参与函数中的局部变量一起定义

例如:

```
fun(a, n)
{
    int a[], n, i;
    :
}
```

上述错误是将函数的形参和函数的局部变量混淆了。函数形参的说明,应该在函数首部进行。而函数内部的局部变量应该在函数体的首部。上述程序应改成:

```
fun(a, n)
int a[], n;
{
    int i;
    :
}
```

或改成:

```
fun(int a[], int n)
{
    int i;
    :
}
```

2. 未对调用的函数作应有的说明

例如:

```
main()
{
    printf("%f", fun(5) );
    :
}
float fun(float x)
{
    :
}
```

此程序编译时将提示出错信息,其原因是 fun 函数是返回值为实型的函数,并且在 main() 函数之后定义,而调用 fun 函数时并未对其进行说明,所以出现错误。有两种改正的方法:一是在函数调用之前用 float fun(float x); 进行提前说明;其二是将 fun 函数的定义放在 main() 函数的前面。

3. 误认为形参值的变化会影响实参的值

例如:

```
main()
{
    int a=5, b=9;
    swap(a, b);
    printf("a=%d, b=%d\n", a, b);
}
swap(x, y)
int x, y;
{ int t;
  t=x; x=y; y=t;
}
```

上述程序原想通过调用 swap 函数交换 a、b 的值,输出结果知 a 和 b 的值并未进行交换,形参值改变并不能改变实参的值。

4. 实参和形参类型的不一致

例如:

```
main()
{
    int a=5, b=9;
    swap(a, b);
    printf("a=%d, b=%d\n", a, b);
}
swap(int *x, int *y)
{
```

```

    int t;
    t = *x;  *x = *y;  *y = t;
}

```

上面程序调用 swap 函数用于交换两个实参变量的值, swap 函数本身没有错误, 但是调用 swap 函数时, 实参 a 和 b 是整型变量, 而形式参数 x 和 y 却是指针变量, 程序编译不会提示错误信息, 但运行后得不到正确的结果。上述程序中, 调用 swap 函数应改成 swap(&a, &b);, 这样才正确。

5. 形参和实参的个数不匹配

例如:

```

float fun(float x, float y)
{
    :
}
main()
{
    printf("%f", fun(6) );
    :
}

```

以上程序中 fun 函数的形参有2个, 而调用时实参却只用了1个。

6. 函数的返回值与期望的不一致

例如:

```

void strlink(char *s1, char *s2)
{
    char *p1=s1, *p2=s2;
    while (*p1) p1++;
    while(*p2) *p1++ = *p2++;
    *p1 = '\0';
}
main()
{
    char str[300]= "abc";
    printf("%s\n", strlink(str, "12345"));
}

```

本例中的主函数期望从 strlink 函数的返值得到一个指向字符串的指针, 而函数 strlink 返回的返回值类型是 void, 即不返回值。可以把 main 函数改成这样:

```

main()
{
    char str[300]= "abc";
    strlink(str, "12345");
    printf("%s\n", str);
}

```



```
}

```

11.4.7 使用指针变量容易出现的错误

1. 定义指针变量时概念混淆

C语言中指针变量的定义有时较为复杂,常常造成定义变量和使用变量不一致。例如:

```
int (*p)(), *q();
```

这里 *p* 是一个指向函数的指针变量,而 *q* 是一个返回值为指针的函数,这在程序设计中较容易混淆。分析复杂的指针变量定义,通常是根据运算符的优先级和结合性来确定的。下面举几个例子:

```
(1)int (*p)[5];
```

①*p* 是一个指针;

②该指针指向一个有5个元素的数组;

③该数组的每个元素是整型(int)类型,即 *p* 是行指针。

```
(2)int *p[5];
```

①*p* 是一个数组;

②*p* 的每个元素都是指针变量;

③*p* 的每个元素都可以指向一个整型变量,即 *p* 是指针数组。

```
(3)int (*p())[5];
```

①*p* 是一个函数;

②该函数返回一个指针;

③该指针指向一个有5个元素的数组;

④该数组的每个元素是整型(int)类型。

2. 混淆不同类型的指针

若有以下定义:

```
int i, *p1=&i;
```

```
float *p2;
```

则赋值语句 *p2=p1*; 是非法的。

3. 混淆指针定义语句中的 * 号和执行语句中的 * 号。

设有以下说明语句:

```
int i, *p;
```

则 **p=&i*; 是非法的。

4. 指针变量未初始化

例如:

```
char *s;
```

```
strcpy(s, "abcdef");
```

语句 *char *s*; 只定义 *s* 是一个指向字符的指针,并没有赋初值。使用指向不确定的指针,在编译时会出现:“NULL pointer assignment”的警告信息。正确的用法是:

```
char str[100]; *s=&str;
```

```
strcpy(s, "abcdef");
```

11.4.8 其他常见错误

1. 混淆字符和字符串

C 语言中的字符常量是由一对单引号括起来的单个字符;而字符串常量是用一对双引号括起来的字符序列。字符常量存放在字符型变量中,而字符串常量可以存放在字符型数组中。例如:

```
char ch;
```

则赋值语句 `ch="A";` 是非法的,字符常量不能用双引号。

2. 不该有空格的地方加了空格

例如,在用 `/* ... */` 对 C 程序中的任何部分作注释时, `/` 与 `*` 之间不应当有空格。又如,在关系运算符 `<=`、`>=`、`==` 和 `!=` 以及逻辑运算符 `&&`、`||`, 两个符号之间也不允许有空格。

3. 混淆结构体类型名和结构体成员名

若定义了以下结构体类型 `booktp`:

```
struct booktp
{
    char name[60];    /* 书名 */
    char author[30];  /* 作者 */
    float price;      /* 价格 */
    struct datetp
    {
        unsigned year;
        unsigned month;
    } pubday;         /* 出版日期 */
} book;
```

赋值语句: `book. datetp. year=2000;` 是非法的,正确的赋值语句是: `book. pubday. year=2000;`

4. 字符串中的一些特殊字符使用错误

例如,有如下语句:

```
file=fopen("c:\temp\abc.txt", "r");
```

目的是打开 C 盘中 TEMP 目录中的 ABC.TXT 文件,这里“\”后面紧接的是“t”将被视为转义字符‘\t’,DOS 将认为它是不正确的文件名而拒绝接受,正确的写法应为:

```
file=fopen("c:\\temp\\abc.txt", "r");
```

类似的字符还有:‘\’、‘\’。

5. 在使用完文件后,没有关闭已打开的文件

多次打开而不随时关闭暂时不使用的文件就会造成文件不够用。有时系统会自动地关闭一些文件,但可能会造成数据的丢失。因此,必须注意随时将暂时不用的文件关闭。

6. 文件打开方式与操作方式不一致

例如下面的程序段:

```
fp=fopen("abc.txt", "r")
```

```

ch=fgetc(fpr);
while(ch!=EOF)
{
    fputc(ch ^ 3f8,fp);
    ch=fgetc(fp)
}

```

在上述程序段中,打开文件是以 "r" 只读方式打开的,而在文件操作时却既要进行读操作又要进行写操作,显然是不允许的。

C 语言程序的调试是程序设计不可缺少的一个重要环节。初学者学好程序调试,就向开发应用软件迈出了关键性的一步。仔细地阅读和实践附录 C 的有关内容,对熟练掌握 Turbo C 2.0 集成开发环境的使用,大有益处。

11.5 小结

本章主要介绍在 Turbo C 2.0 集成开发环境下调试程序,以及 Turbo C 程序的常见错误。应重点掌握以下几个方面的内容:

1. 熟练掌握 Turbo C 2.0 集成开发环境的使用,包括编辑窗口、信息窗口、监视窗口中使用快捷键,阅读理解提示信息,熟悉各项菜单的意义。
2. 掌握调试程序的基本要领,包括纠正编译错误和连接错误,利用单步调试、跳跃跟踪、观察监视窗口表达式的值变化情况以及充分利用运算窗口功能来查找和纠正程序的逻辑错误。
3. 了解 Turbo C 程序中常见错误的类型,使得在编写程序时尽量少出错误。

习 题

- 11.1 编辑、运行、调试[例4.2]、[例5.11]、[例6.19]、[例7.17]、[例7.21]。
- 11.2 编写程序,并调试运行习题4.43、习题4.51、习题5.48和习题5.55。
- * 11.3 设计一个计算器程序,具有加、减、乘、除四则运算功能。用户输入一个算术表达式,程序输出表达式的值,如输入: $-1.23 * (-4.5 + 6.78 / (9.12 * 1.29 - 8))$, 则输出: 3.3199。按 Esc 键退出程序。

附录 A C 语法摘要

1. 标识符、常量与变量

标识符是以字母或下划线“_”开头的,由字母、数字和下划线组成的字符序列,用来标识变量、常量、数据类型、函数等。C 语言中,大、小写字母分别代表不同的标识符,也就是说,C 语言是区分大小写的。在 Turbo C 中,对一个标识符识别它的前 32 个字符,即标识符的有效长度为 32 个字符。

在 C 语言中,有一些标识符是具有固定意义的,称为关键字,如 `int`、`switch`、`typedef` 等。在编写程序时应注意不要使用这些标识符作为用户标识符。

在 C 系统中,一些标识符被系统所定义,称为预定义标识符,如系统库函数名、系统定义的宏等。它们在系统头文件中定义,一旦程序中包含了头文件,则在这些头文件中定义的标识符就起作用。

常量是在程序运行时值不发生变化的量。包括:

(1) 整型常量

- ① 十进制常数;
- ② 八进制常数(以 0 开头的数字序列);
- ③ 十六进制常数(以 0x 开头的数字序列);
- ④ 长整型数(在数字后加字符 L 或 l)。

(2) 浮点数常量

- ① 小数形式;
- ② 指数形式。

(3) 字符常量

字符常量是用单引号“'”括起来的单个字符。可以使用转义字符。

(4) 字符串常量

字符串常量是用双引号“”括起来的字符序列。

(5) 符号常量

符号常量是用 `#define` 定义的无参数宏。

在程序执行过程中,值可以发生变化的数据称为变量。在 C 语言程序中,用户可以根据需要,用任意一个合法的标识符(关键字和预定义标识符除外)来命名变量。

2. 表达式和运算符

表达式是 C 语言中最基本的计算成分,它是由各种操作数或者操作数和运算符构成的序列。这些操作数可以是变量、常量以及函数的调用等等。

C 语言具有十分丰富的运算符。C 语言的编译系统把控制语句和输入输出以外的所有基本操作都作为运算符处理。C 语言的运算符可以分为以下几类:

- (1) 算术运算符: `+`、`-`、`*`、`/`、`%`
- (2) 关系运算符: `>`、`>=`、`==`、`<=`、`<`、`!=`
- (3) 逻辑运算符: `&&`、`||`、`!`

- (4)位运算符: &、|、^、~、<<、>>
 (5)赋值运算符: =、*=、+=、-=、/=、%=、>>=、<<=、&=、^=、|=
 (6)条件运算符: ?:
 (7)逗号运算符: ,
 (8)长度运算符: sizeof()
 (9)指针运算符: *(指针运算符)、&(取地址运算符)
 (10)结构体运算符: .(结构体成员运算符)、->(指向结构体成员运算符)
 (11)增减运算符: ++(自增)、--(自减)
 (12)其他运算符: 如下标、括号、类型转换及符号(正、负)运算符等

运算符有两个特性:优先级别和结合方向。优先级别反映运算符在表达式中操作的先后次序(参考第二章);结合方向反映同一优先级别的运算符在表达式中操作的组织方向。C 语言中规定了两种结合方向:一种是“左结合性”,即按从左到右的方向进行运算;另一种是“右结合性”,即按从右到左的方向进行运算。

不同运算符下表达式值的数据类型不尽相同,如由算术运算符构成的表达式值的数据类型由操作数中的最高精度类型确定;指针运算符(*)构成的表达式值的数据类型由其操作数(指针)的指针数据类型决定等等。

3. 变量

C 语言规定,在程序中所有变量都必须在使用前进行定义。变量定义的一般形式如下:

存储类别说明符 类型说明符 变量表列

其中,存储类别说明符有:auto、static、extern、register 等。

类型说明符有:char、short、int、long、unsigned、float、double 等

变量表列中各变量之间用逗号“,”分开。

可以用类型限定符 const、volatile 说明变量在程序中是不能显式改变的或者易变的。

类型说明符可以用 typedef 定义的类型名。

各种变量的定义形式如下:

(1)简单变量

定义形式:存储类别说明符 类型说明符 变量表列

(2)数组

定义形式:存储类别说明符 类型说明符 变量表列[常量表达式]…[常量表达式]

(3)结构体变量

定义形式:

①struct {结构说明表} 变量表列

②struct 标识符 {结构说明表} 变量表列

③struct 标识符 {结构说明表}

struct 标识符 变量表列

(4)共同体变量

定义形式:

①union {结构说明表} 变量表列

②union 标识符 {结构说明表} 变量表列

③union 标识符 {结构说明表}

union 标识符 变量表列

(5)枚举类型变量

定义形式:

①enum {结构说明表} 变量表列

②enum 标识符 {结构说明表} 变量表列

③enum 标识符 {结构说明表}

enum 标识符 变量表列

C 程序中,每个变量都有其作用范围,遵循作用域规则。在使用变量前,应首先对变量进行初始化。系统自动为静态变量或外部变量进行初始化工作,使它们具有初值零(数值型变量)或空(字符型数据)。对于自动变量或寄存器变量,若在程序中未初始化,则其初值为不可预测的值。

4. 语句

和其他高级语言一样,C 语言的语句用来向计算机系统发出操作指令。在 C 语言中只有“可执行语句”,而没有“非执行语句”。C 语言的语句可以分为以下五类:

(1)控制语句:完成一定的控制功能的语句。C 语言只有 9 种控制语句,它们是:

①if(条件语句)

当满足某给定条件时,执行一组语句,否则,执行另外一组语句。

②for(循环语句)

这是 C 语言中使用最广泛、最灵活的循环语句。其一般形式为:

for(表达式 1;表达式 2;表达式 3)

{循环体语句}

③while(循环语句)

先判断循环结束条件:如果条件成立,则执行循环体语句,否则,跳出循环体。很明显这种循环语句实现的是一种“当型”循环。

④do while(循环语句)

先执行循环体语句,再进行循环结束条件的判断。它属于“直到型”循环。

⑤continue(结束本次循环语句)

⑥break(中止执行 switch 或循环语句)

⑦switch(开关语句)

⑧goto(转向语句)

⑨return(从函数返回语句)

(2)表达式语句:由一个表达式加一个分号构成的语句。

(3)空语句:空语句就是一个分号,它不完成任何工作。

(4)复合语句:用 {} 将一些语句括起来组成的语句。在语法上,复合语句的作用与一个简单语句相同。

5. 函数定义

函数是 C 语言的基本组成单元。一个 C 程序是由一个或多个函数组成的,其中有且仅有

一个主函数 main。每个函数相当于一个程序模块,有它自己的功能。C 语言规定:所有的函数都是平行的,因此函数不能嵌套定义。函数可以嵌套调用。C 语言的函数允许直接或间接地调用自己,这称为函数的递归调用。

C 语言的函数按其形式可分为三类:无参函数、有参函数和空函数。它们的定义形式分述如下:

(1) 无参函数

定义形式:

存储类别 函数类型 函数名()

{说明部分

函数体}

(2) 有参函数

定义形式:

存储类别 函数类型 函数名(形参表列)

{说明部分

函数体}

(3) 空函数

定义形式:

存储类别 函数类型 函数名(形参表列)

{ }

在一个 C 源程序中,函数名具有全局作用域,与全局变量类似,它只有 extern 和 static 两种存储类别。函数调用时,函数调用表达式值的数据类型由函数类型指定,函数类型即为函数返回值类型。形参表列包含形参变量的说明,即包含形参变量的数据类型说明。

6. 编译预处理

C 语言提供编译预处理功能,这是它与其他高级语言的一个重要区别。在 C 编译系统对程序进行通常的编译之前,先对程序中一些特殊的命令进行“预处理”,然后将预处理的结果与源程序一起再进行通常的编译,以得到目标代码。C 语言的编译预处理功能有三种:

(1) 宏定义

① 不带参数的宏定义

#define 标识符 字符串

系统在进行通常意义上的编译之前,把此命令行之后出现的该标识符都用此定义行中的字符串代替,即进行所谓编译预处理。然后再进行通常意义上的编译。

② 带参数宏定义

在标识符替换的同时,又进行参数替换。其定义的一般形式为:

#define 宏名(参数表) 字符串

(2) 文件包含

所谓“文件包含”处理是指一个源文件可将另外一个源文件的全部内容包含进来。定义的一般形式为:

#include <文件名>

或: #include "文件名"

使用尖括号与使用双引号的区别在于:使用尖括号引起系统在系统规定的标准路径上查找文件;而使用双引号则是首先在当前目录中查找文件,如果找不到,则继续按系统规定的标准路径查找文件。所以,使用双引号要安全可靠一些。一般地,如果需要使用库文件,则用尖括号;如果需要使用用户自己定义的文件则使用双引号。

(3) 条件编译

一般情况下,源程序中所有的行都参加编译。但有时希望对其中一部分内容只在满足一定的条件下才进行编译,也就是对一部分内容指定编译的条件,这就是所谓“条件编译”。条件编译有以下几种形式:

- ① #ifdef
- ② #ifndef
- ③ #if

7. 文件的输入与输出

C 语言没有输入输出(I/O)语句,其 I/O 功能都是通过函数来实现的。C 语言提供丰富的输入输出函数,缓冲文件系统的函数原型放在头文件 `stdio.h` 中,非缓冲文件系统的头文件放在 `io.h` 中。在程序中如果需要使用它们,可以用文件包含命令将这个文件包含到其中,通过对这些函数的调用,实现文件操作。

附录 B 数值系统

1. 数值系统介绍

日常生活中我们使用十进制来表示数值,在此基础上进行各种算术运算及编码操作。数值系统规定了数值的表示方法,并在此基础上衍生了顺序(大小)及运算规则。以十进制数 3456 为例,我们来看数值系统是如何表示一个数值的。

在十进制数值系统中,我们知道:

$$(3456)_{10} = 3 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$$

其中,10 称为十进制数值系统的基数,3、4、5、6 称为十进制数值系统的数码,十进制数值系统中数码还有:0、1、2、7、8、9。在数值系统中采用按位记数法,每个数位有一个不同的位值:10⁰ 为个位,10¹ 为十位,10² 为百位,10³ 为千位等等,因此一个十进制数中的每一个数码代表的数值由数码和位值共同决定,如 3456 中 3 代表 3000,4 代表 400 等等。一个十进制数的值的等于每个数码代表的数值之和。

一般地,在十进制数值系统中,对于一个十进制数 $D_n D_{n-1} \cdots D_1 D_0, D_{-1} D_{-2} \cdots D_{-m+1} D_{-m}$, 有:

$$D_n D_{n-1} \cdots D_1 D_0, D_{-1} D_{-2} \cdots D_{-m+1} D_{-m} = D_n \times 10^n + D_{n-1} \times 10^{n-1} + \cdots + D_1 \times 10^1 + D_0 \times 10^0 \\ + D_{-1} \times 10^{-1} + D_{-2} \times 10^{-2} + \cdots + D_{-m+1} \times 10^{-m+1} + D_{-m} \times 10^{-m}$$

在计算机中,数值是用二进制表示的。在二进制数值系统中,基数为 2,数码只有两个:0 和 1,数位有:个位(2⁰)、二位(2¹)、四位(2²)等等。一般地,对于一个二进制数 $B_n B_{n-1} \cdots B_1 B_0, B_{-1} B_{-2} \cdots B_{-m+1} B_{-m}$, 有:

$$B_n B_{n-1} \cdots B_1 B_0, B_{-1} B_{-2} \cdots B_{-m+1} B_{-m} = B_n \times 2^n + B_{n-1} \times 2^{n-1} + \cdots + B_1 \times 2^1 + B_0 \times 2^0 + B_{-1} \times 2^{-1} \\ + B_{-2} \times 2^{-2} + \cdots + B_{-m+1} \times 2^{-m+1} + B_{-m} \times 2^{-m}$$

$$\text{如: } (1101)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (13)_{10}$$

可以看到,对于稍大一点的数值来说,二进制数比等值的十进制数要长得多,这使得程序员使用它们不方便,且容易出错,因此我们常使用八进制和十六进制数值系统来简化二进制数表示。

八进制数值系统的基数为 8,数码范围为 0~7,数位有:个位(8⁰)、八位(8¹)、十六位(8²)等等。一般地,对于一个八进制数 $O_n O_{n-1} \cdots O_1 O_0, O_{-1} O_{-2} \cdots O_{-m+1} O_{-m}$, 有:

$$O_n O_{n-1} \cdots O_1 O_0, O_{-1} O_{-2} \cdots O_{-m+1} O_{-m} = O_n \times 8^n + O_{n-1} \times 8^{n-1} + \cdots + O_1 \times 8^1 + O_0 \times 8^0 \\ + O_{-1} \times 8^{-1} + O_{-2} \times 8^{-2} + \cdots + O_{-m+1} \times 8^{-m+1} + O_{-m} \times 8^{-m}$$

$$\text{如: } (3456)_8 = 3 \times 8^3 + 4 \times 8^2 + 5 \times 8^1 + 6 \times 8^0 = (1838)_{10}$$

十六进制数值系统的基数为 16,数码范围为 0~9、A~F,数码 A~F 分别对应十进制的 10~15,数位有:个位(16⁰)、十六位(16¹)、二百五十六位(16²)等等。一般地,对于一个十六进制数 $H_n H_{n-1} \cdots H_1 H_0, H_{-1} H_{-2} \cdots H_{-m+1} H_{-m}$, 有:

$$H_n H_{n-1} \cdots H_1 H_0, H_{-1} H_{-2} \cdots H_{-m+1} H_{-m} = H_n \times 16^n + H_{n-1} \times 16^{n-1} + \cdots + H_1 \times 16^1 + H_0 \times 16^0 \\ + H_{-1} \times 16^{-1} + H_{-2} \times 16^{-2} + \cdots + H_{-m+1} \times 16^{-m+1} + H_{-m} \times 16^{-m}$$

$$\text{如: } (3456)_{16} = 3 \times 16^3 + 4 \times 16^2 + 5 \times 16^1 + 6 \times 16^0 = (13398)_{10}$$

在以上数的表示中再冠以正负号(+、-)就得到了完整的数值表示。

2. 二进制数与八进制和十六进制数之间的转换

八进制和十六进制数与二进制数有着简洁的对应关系,可以用来简化冗长的二进制数,这是因为八进制和十六进制数值系统的基数($8=2^3$, $16=2^4$)均为 2(二进制的基数)的幂次的缘故。考察 12 位的二进制数 1101011.01011 以及与之相等的八进制数和十六进制数可以看出其简单的转换方法:

$$\begin{aligned}
 (1101011.01011)_2 &= (001)_2 \times 8^2 + (101)_2 \times 8^1 + (011)_2 \times 8^0 + (010)_2 \times 8^{-1} + (110)_2 \times 8^{-2} \\
 &= 1 \times 8^2 + 5 \times 8^1 + 3 \times 8^0 + 4 \times 8^{-1} + 6 \times 8^{-2} = (153.46)_8 \\
 &= (0110)_2 \times 16^1 + (1011)_2 \times 16^0 + (0101)_2 \times 16^{-1} + (1000)_2 \times 16^{-2} \\
 &= 6 \times 16^1 + B \times 16^0 + 5 \times 16^{-1} + 8 \times 16^{-2} = (6B.58)_{16}
 \end{aligned}$$

从上式可以看出,二进制数转换成八进制,只需将二进制数从小数点位置往左及往右按每连续三位为一组,并将每组三位二进制数用等值的八进制数码写出即可;八进制数转换成二进制数同样简单:将每个八进制数码用等值的三位二进制数写出,顺序将这些二进制数拼在一起就是与原八进制数等值的二进制数了。二进制数与十六进制数的转换也是类似的,只是与十六进制数码相对应的是四位二进制数。

表 B.1 八、十六进制数码转换二进制参考表

八进制数码	十六进制数码	等值的二进制数	等值的十进制数
0	0	0	0
1	1	1	1
2	2	10	2
3	3	11	3
4	4	100	4
5	5	101	5
6	6	110	6
7	7	111	7
	8	1000	8
	9	1001	9
	A	1010	10
	B	1011	11
	C	1100	12
	D	1101	13
	E	1110	14
	F	1111	15

3. 二、八、十六进制数与十进制数之间的转换

我们已经习惯了使用十进制数,因此经常将二、八和十六进制数与十进制数之间进行转换。二、八和十六进制数与十进制数之间的转换根据前面介绍的公式进行:

$$\begin{aligned} B_n B_{n-1} \cdots B_1 B_0, B_{-1} B_{-2} \cdots B_{-m+1} B_{-m} &= B_n \times 2^n + B_{n-1} \times 2^{n-1} + \cdots + B_1 \times 2^1 + B_0 \times 2^0 + B_{-1} \times 2^{-1} \\ &\quad + B_{-2} \times 2^{-2} + \cdots + B_{-m+1} \times 2^{-m+1} + B_{-m} \times 2^{-m} \\ &= D_i D_{i-1} \cdots D_1 D_0, D_{-1} D_{-2} \cdots D_{-j+1} D_{-j} \end{aligned}$$

$$\begin{aligned} O_n O_{n-1} \cdots O_1 O_0, O_{-1} O_{-2} \cdots O_{-m+1} O_{-m} &= O_n \times 8^n + O_{n-1} \times 8^{n-1} + \cdots + O_1 \times 8^1 + O_0 \times 8^0 + \\ &\quad O_{-1} \times 8^{-1} + O_{-2} \times 8^{-2} + \cdots + O_{-m+1} \times 8^{-m+1} + \\ &\quad O_{-m} \times 8^{-m} \\ &= D_i D_{i-1} \cdots D_1 D_0, D_{-1} D_{-2} \cdots D_{-j+1} D_{-j} \end{aligned}$$

$$\begin{aligned} H_n H_{n-1} \cdots H_1 H_0, H_{-1} H_{-2} \cdots H_{-m+1} H_{-m} &= H_n \times 16^n + H_{n-1} \times 16^{n-1} + \cdots + H_1 \times 16^1 + H_0 \times 16^0 + \\ &\quad H_{-1} \times 16^{-1} + H_{-2} \times 16^{-2} + \cdots + H_{-m+1} \times 16^{-m+1} + \\ &\quad H_{-m} \times 16^{-m} \\ &= D_i D_{i-1} \cdots D_1 D_0, D_{-1} D_{-2} \cdots D_{-j+1} D_{-j} \end{aligned}$$

根据上述公式的描述,在将二、八和十六进制数转换为十进制数时,是将一个二、八和十六进制数中的每个数码的十进制数值乘以其位值,然后求出它们的和即为相应的十进制数。

清晰起见,下面以表格形式例举二、八和十六进制数转换为十进制数:

表 B.2 将二进制数 100101 转换为十进制数

位值	32	16	8	4	2	1	
数码	1	0	0	1	0	1	
乘积	32	0	0	4	0	1	
总和	32	+		4	+	1	=37

表 B.3 将八进制数 507 转换为十进制数

位值	64	8	1	
数码	5	0	7	
乘积	320	0	7	
总和	320	+	7	=327

表 B.4 将十六进制数 A3B 转换为十进制数

位值	256	16			1		
数码	A	3			B		
乘积	2560	48			11		
总和	2560	+	48	+	11	=2619	

十进制数转换为二、八、十六进制数是上述过程的逆过程:

$$D_i D_{i-1} \cdots D_1 D_0, D_{-1} D_{-2} \cdots D_{-j+1} D_{-j}$$

$$= (\cdots (B_n \times 2 + B_{n-1}) \times 2) + \cdots + B_1) \times 2 + B_0 + 2^{-1} \times (B_{-1} + 2^{-1} \times (B_{-2} + \cdots + 2^{-1} \times (B_{-m+1} + 2^{-1} \times B_{-m}) \cdots))$$

$$= B_n \times 2^n + B_{n-1} \times 2^{n-1} + \cdots + B_1 \times 2^1 + B_0 \times 2^0 + B_{-1} \times 2^{-1} +$$

$$B_{-2} \times 2^{-2} + \cdots + B_{-m+1} \times 2^{-m+1} + B_{-m} \times 2^{-m}$$

$$= B_n B_{n-1} \cdots B_1 B_0, B_{-1} B_{-2} \cdots B_{-m+1} B_{-m}$$

$$D_i D_{i-1} \cdots D_1 D_0, D_{-1} D_{-2} \cdots D_{-j+1} D_{-j}$$

$$= (\cdots (O_n \times 8 + O_{n-1}) \times 8) + \cdots + O_1) \times 8 + O_0 + 8^{-1} \times (O_{-1} + 8^{-1} \times (O_{-2} + \cdots + 8^{-1} \times (O_{-m+1} + 8^{-1} \times O_{-m}) \cdots))$$

$$= O_n \times 8^n + O_{n-1} \times 8^{n-1} + \cdots + O_1 \times 8^1 + O_0 \times 8^0 + O_{-1} \times 8^{-1} + O_{-2} \times 8^{-2} + \cdots +$$

$$O_{-m+1} \times 8^{-m+1} + O_{-m} \times 8^{-m}$$

$$= O_n O_{n-1} \cdots O_1 O_0, O_{-1} O_{-2} \cdots O_{-m+1} O_{-m}$$

$$D_i D_{i-1} \cdots D_1 D_0, D_{-1} D_{-2} \cdots D_{-j+1} D_{-j}$$

$$= (\cdots (H_n \times 16 + H_{n-1}) \times 16 + \cdots + H_1) \times 16 + H_0 + 16^{-1} \times (H_{-1} + 16^{-1} \times (H_{-2} + \cdots + 16^{-1} \times (H_{-m+1} + 16^{-1} \times H_{-m}) \cdots))$$

$$= H_n \times 16^n + H_{n-1} \times 16^{n-1} + \cdots + H_1 \times 16^1 + H_0 \times 16^0 + H_{-1} \times 16^{-1} +$$

$$H_{-2} \times 16^{-2} + \cdots + H_{-m+1} \times 16^{-m+1} + H_{-m} \times 16^{-m}$$

$$= H_n H_{n-1} \cdots H_1 H_0, H_{-1} H_{-2} \cdots H_{-m+1} H_{-m}$$

以上三式暗示了将十进制数转换成二、八、十六进制数的方法。以十进制数转换为二进制数为例,其整数部分可以采用除 2(二进制的基数)取余法,即将已知十进制整数(及其除 2 的商)反复除以 2,在每次相除之后,若余数为 1,则对应于二进制数码 1,否则为数码 0,直到除得的商为 0。首次除法得到的余数是二进制数整数部分的个位,最末一次除法所得的余数是二进制数整数部分的最高位等等。其小数部分可以采用乘 2 取整法,即将已知十进制小数(及其乘 2 的积的小数部分)反复乘以 2,在每次相乘之后,若整数部分为 1,则对应于二进制数码 1,否则为数码 0,直到乘得的积没有小数部分或达到精度要求为止。

举例说明将十进制数 215.6 转换为二进制数。

整数部分 215:

$215 = 107 \times 2 + 1$	余数为 1	$B_0 = 1$
$107 = 53 \times 2 + 1$	余数为 1	$B_1 = 1$
$53 = 26 \times 2 + 1$	余数为 1	$B_2 = 1$
$26 = 13 \times 2 + 0$	余数为 0	$B_3 = 0$
$13 = 6 \times 2 + 1$	余数为 1	$B_4 = 1$
$6 = 3 \times 2 + 0$	余数为 0	$B_5 = 0$
$3 = 1 \times 2 + 1$	余数为 1	$B_6 = 1$
$1 = 0 \times 2 + 1$	余数为 1	$B_7 = 1$

十进制整数 215 等于二进制整数 $B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0 = 11010111$

小数部分 0.6:

$0.6 \times 2 = 0.2 + 1$	整数部分为 1	$B_{-1} = 1$
$0.2 \times 2 = 0.4 + 0$	整数部分为 0	$B_{-2} = 0$

$$0.4 \times 2 = 0.8 + 0 \quad \text{整数部分为 } 0 \quad B_{-3} = 0$$

$$0.8 \times 2 = 0.6 + 1 \quad \text{整数部分为 } 1 \quad B_{-4} = 1$$

$$0.6 \times 2 = \dots (\text{循环})$$

可以看到有限位十进制小数未必能转换成有限位二进制小数,此时取若干位满足精度即可。若取 4 位小数,十进制小数 $0.6 \approx 0.B_{-1}B_{-2}B_{-3}B_{-4} = 0.1001$ 。从而得到十进制数 215.6 转换成的(近似)二进制数 11010111.1001。

附录 C Turbo C 2.0 集成开发环境的使用

1. Turbo C 2.0 基本配置要求

Turbo C 2.0 可运行于带有硬盘或网络无盘工作站的 IBM PC 系列兼容微机上。操作系统要求 DOS 2.0 或更高版本。可在任何彩、单色 80 列监视器上运行。

2. Turbo C 2.0 文件内容简介

以下为 Turbo C 2.0 的主要文件：

INSTALL.EXE	安装程序文件
TC.EXE	集成开发环境
TCINST.EXE	集成开发环境的配置程序
TCHELP.TCH	帮助文件
TCHELP.COM	读取 TCHHELP.TCH 的驻留程序
TCCONFIG.EXE	配置开发环境文件
MAKE.EXE	工程管理工具
TCC.EXE	Turbo C 命令行编译
TLINK.EXE	Turbo C 系列连接器
TLIB.EXE	Turbo C 系列库管理工具
README	Turbo C 说明文件
HELPME.DOC	一般性问题及解答
CPP.EXE	Turbo C 预处理程序
TOUCH.COM	日期和时间的更改程序
C0?.OBJ	不同模式启动代码
C?.LIB	不同模式运行库
GRAPHICS.LIB	图形库
EMU.LIB	8087 仿真库
FP87.LIB	8087 库
*.H	Turbo C 头文件
*.BGI	不同显示器图形驱动程序
*.C	Turbo C 范例源程序

其中,上面的? 分别为:

T	Tiny(微模式)
S	Small(小模式)
C	Compact(紧凑模式)
M	Medium(中模式)
L	Large(大模式)
H	Huge(巨模式)

3. Turbo C 2.0 的安装和启动

(1) Turbo C 2.0 的安装

将 1# 盘插入 A 驱动器中,在 DOS 下键入:

A:\>INSTALL

此时屏幕上显示如下三种选择:

```

Install Turbo C on a Hard Drive(硬盘上安装 Turbo C 系统)
Update Hard Drive Copy Turbo C 1.5 to Turbo C 2.0
(把硬盘上的 Turbo C 1.5 升级成 Turbo C 2.0 版本)
Install Turbo C on a Floppy Drive (软盘上安装 Turbo C 系统)
  
```

一般选择第一种方式进行安装,在安装过程中顺序插入各个软盘,安装完成后将在 C 盘根目录下建立一个 TC 子目录,TC 下还建立了两个子目录 LIB 和 INCLUDE,C:\TC\LIB 子目录中存放库文件,C:\TC\INCLUDE 子目录中存放所有头文件。

若是复制已安装好的 Turbo C 2.0 系统,还应设置好菜单栏中 Option/Directories 的各项内容与所复制的目录一致。

(2) Turbo C 2.0 的启动

只要在 TC 子目录下键入 TC 并回车即可进入 Turbo C 2.0 集成开发环境。

4. Turbo C 2.0 的主屏幕

进入 Turbo C 2.0 集成开发环境中后,将显示如下 Turbo C 的主屏幕:

File Edit Compile Project Option Debug Break/Watch								}	菜单栏
Edit									
Line 1	Col 1	Insert	Indent	Tab Fill	Unindent	C:NONAME.C		}	编辑窗口
Message								}	信息窗口
F1-Help F5-Zoom F6-Switch F7-Trace F8-Step F9-Make F10-Menu								}	快捷键提示

其中顶上一行为 Turbo C 2.0 菜单栏,中间窗口为编辑窗口,接下来是信息窗口,最底下一行为快捷键提示行。这四个窗口构成了 Turbo C 2.0 的主屏幕,程序的编辑、编译、调试以及运行都将在主屏幕中进行。下面详细介绍主屏幕中的各项内容。

5. 编辑窗口

编辑窗口的顶部有一个编辑状态行,它给出了正在编辑文件的有关信息:

Line 1	当前光标所在的行号
Col 1	当前光标所在的列号
Insert	当前编辑程序处于插入状态,按 Insert 键转为改写状态
Indent	自动缩进开关为开,此时换行,光标自动地与上一行的行首对齐 可用 Ctrl+OI 控制其开关
Tab	制表开关,可用 Ctrl+OT 控制开关
Fill	当 TAB 开关开时,编辑程序在每一行的开始填以适当的制表符和空格 可用 Ctrl+OF 来控制开关
Unindent	当光标处于某行的第一个非空字符时,退格键将使光标左移一级 可用 Ctrl+OU 来控制开关
*	当文件已修改,但未保存时,*号将出现在文件名的前面

C:\NONAME.C 表明编辑文件在 C 盘当前目录,文件名为 NONAME.C

编辑窗口用于编辑源程序文件,所用的编辑命令见表 C.1。

表 C.1

分类	命令组合键	功能
移动光标	↑ ↓ ← → CTRL+← CTRL+→	光标上移一行 光标下移一行 光标左移一个字符 光标右移一个字符 光标左移一个单词 光标右移一个单词
快速移动光标命令	HOME END CTRL+HOME CTRL+END CTRL+PageUp CTRL+PageDown CTRL+QB CTRL+QK CTRL+QP CTRL+KN CTRL+QN	光标移到行首 光标移到行尾 光标移到窗口左上角 光标移到窗口右下角 光标移到文件首 光标移到文件尾 光标移到块首 光标移到块尾 光标移到上次位置 n=0,1,2,3 设置书签位置 n=0,1,2,3 光标移到对应的书签位置
插入与删除	INSERT CTRL+N CTRL+Y CTRL+QY BACKSPACE DEL CTRL+T CTRL+QL	插入状态/改写状态转换开关 插入一行 删除光标所在行 删除到行尾 删除光标左边的一个字符 删除光标处的一个字符 删除光标右边的一个单词 恢复原行

(续表)

分类	命令组合键	功能
块操作命令	CTRL+KB CTRL+KK CTRL+KT CTRL+KP CTRL+KC CTRL+KY CTRL+KH CTRL+KV CTRL+KR CTRL+KW CTRL+KU CTRL+KI	标记块头 标记块尾 标记单词 打印块 复制块插入到光标处 删除块 隐藏/显示块 搬移块插入到光标处 从磁盘上读文件插入到光标处 把块存成磁盘文件 块向左平移 块向右平移
其他	ESC CTRL+QF CTRL+QA CTRL+L F3 ALT+F3 F2 或 CTRL+KS CTRL+OT CTRL+OI CTRL+Q[CTRL+Q]	中止操作 查找 查找并替换 重复上次查找或替换 装入文件到编辑窗口 选取文件并装入到编辑窗口 将当前文件存盘返回编辑 制表模式 自动缩进开关 查找限定符的后匹配符 查找限定符的前匹配符

Turbo C 2.0 的限定符包括以下几种符号:

花括号 { }
尖括号 < >
圆括号 ()
方括号 []
注释符 /* 和 */
双引号 "
单引号 '

6. 信息窗口

在编译和连接程序出现错误时,系统将自动激活信息窗口,并列出每个警告和错误信息,同时用高亮度光条在编辑窗口中标出程序的相应出错位置。光标位于信息窗口时,常用的快捷键有:

F5-Zoom 将活动窗口扩展为整个屏幕,再按一次将恢复原有屏幕尺寸
F6-Switch 切换编辑窗口、信息窗口及监视窗口为活动窗口

7. 监视窗口

在集成开发环境中执行并调试程序时,信息窗口被监视窗口代替。调试程序时,可以把某些变量或表达式加入到监视窗口中,逐行执行一行程序时可以观察到监视窗口中变量及表达式值的变化情况。光标位于监视窗口时,常用的快捷键有:

F5-Zoom	将活动窗口扩展为整个屏幕,再按一次将恢复原有屏幕尺寸
F6-Switch	切换编辑窗口、信息窗口及监视窗口为活动窗口
Insert	向监视窗口增加一个表达式
Delete	从监视窗口中删除一个表达式
Enter	编辑当前的监视表达式

8. 快捷键提示

在主屏幕的最底行是快捷键提示,其各键的意义如下:

F1-Help	打开帮助窗口,提供有关编辑命令的帮助信息
F5-Zoom	将活动窗口扩展为整个屏幕,再按一次将恢复原有屏幕尺寸
F6-Switch	切换编辑窗口、信息窗口及监视窗口为活动窗口
F7-Trace	调试程序时逐行执行程序,遇到用户函数单步跟踪进入用户函数内部
F8-Step	调试程序时逐行执行程序,遇到用户函数不跟踪进入用户函数内部
F9-Make	编译、连接并生成可执行文件(注意:此快捷键并未执行程序)
F10-Menu	切换活动窗口与菜单栏

9. 菜单栏

菜单栏在 Turbo C 2.0 主屏幕顶行,显示下列内容:

File Edit Run Compile Project Options Debug Break/watch

用 Alt 加上某项中第一个字母,就可进入该项的子菜单中,例如按 **[Alt]+[F]** 可进入 File(文件)菜单。以下顺序介绍各菜单项的内容。

(1) File(文件)菜单

该菜单的内容及意义见表 C.2。

表 C.2

子菜单	意义	说 明
Load	装入文件	从磁盘装入一个文件,可用通配符(如 *.C)来进行列表选择。也可装入其他扩展名的文件,只要给出文件名(或只给路径)即可。该项的热键为 F3。
Pick	选取文件	将最近装入编辑窗口的 8 个文件列成一个表让用户选择,选择后将该程序装入编辑窗口。其热键为 Alt+F3。
New	新文件	建立新文件,缺省文件名为 NONAME.C,存盘时可改名。
Save	文件存盘	将编辑窗口中的文件存盘,若文件名是 NONAME.C 时,将询问是否更改文件名存盘。其热键为 F2 或 Ctrl+KS。
Write to	另存为	可由用户给出文件名将编辑窗口中的内容另存为其他文件名,若该文件已存在,则询问要不要覆盖。
Directory	目录	显示目录及目录中的文件,供用户选择。
Change dir	改变目录	显示当前目录,用户可以改变当前目录。
Os shell	暂时退出	暂时用 shell 方式退出 Turbo C 2.0,回到 DOS 提示符下,此时可以运行 DOS 命令,若要回到 Turbo C 2.0 集成开发环境,在 DOS 状态下键入 EXIT,回车。
Quit	退出	退出 Turbo C 2.0,返回到 DOS 操作系统中。其热键为 Alt+X。

(2) Edit(编辑)菜单

按 Alt+E 进入编辑菜单,回车则光标出现在编辑窗口,此时用户可以进行文本编辑。与编辑有关的命令见上述“5. 编辑窗口”。

(3) Run(运行)菜单

该菜单的内容及意义见表 C.3。

表 C.3

子菜单	意义	说 明
Run	执行程序	编译生成目标文件、连接生成可执行文件并立即执行程序。用 Alt+F5 可以切换到用户屏幕,看到程序执行结果。其热键为 Ctrl+F9。
Program reset	程序重启	中止当前程序调试,再次运行时将从头开始。其热键为 Ctrl+F2。
Go to cursor	运行到光标处	调试程序时使用。选择该项可使程序运行到光标所在行。光标所在行必须为一条可执行语句,否则提示错误。其热键为 F4。
Trace into	单步跟踪	调试程序时使用。逐行执行程序,单步跟踪到用户函数内部。其热键为 F7。
Step over	单步跟踪但不进入函数	调试程序时使用。逐行执行程序,遇到用户函数不进入函数内部。其热键为 F8。
User screen	用户屏幕	显示程序运行时在屏幕上的结果。其热键为 Alt+F5。

(4) Compile(编译)菜单

该菜单的内容及意义见表 C.4。

表 C.4

子菜单	意义	说 明
Compile to OBJ	编译生成目标文件	将一个 C 源文件编译生成 OBJ 目标文件,同时显示生成的文件名。其热键为 Alt+F9。
Make EXE file	编译生成目标文件后再连接生成可执行文件	此命令的热键是 F9。此命令生成一个 EXE 的文件。其中, EXE 文件名是下面几项之一: 1. Project/Project name 中指定的项目文件名; 2. 若没有项目文件名,则是 Primary C file 中文件名; 3. 若以上两项都没有指定文件名,则为当前窗口文件名。
Link EXE file	连接生成可执行文件	把当前 OBJ 文件和库文件连接在一起生成 EXE 文件。
Build all	生成所有的目标文件和可执行文件	重新编译项目里的所有文件,再生成 EXE 文件。本命令不作过时检查。所谓过时检查就是:如果目前项目里源文件的时间与目标文件的时间相同或更早,则不对源文件进行编译。
Primary C file	主 C 文件	当在该项中指定了主文件后,在以后的编译中,如没有项目文件名,则编译此项中指定的主 C 文件。
Get info	获取信息	获得有关当前路径、源文件名、源文件字节大小、编译中的错误数目、内存可用空间等信息。

(5)Project(项目)菜单,也称(工程)菜单

该菜单的内容及意义见表 C. 5。

表 C. 5

子菜单	意义	说 明
Project name	项目名称	<p>项目名称是以 .PRJ 为扩展名的文件,其内容包括将要编译、连接的文件名。例如有一个程序由 file1. c、file2. obj(用 C 编写的源程序编译的,或者用汇编语言编写的源程序编译的)、file3. c 组成。要将这 3 个文件编译连接成一个 file. exe 的执行文件,可以先建立一个名为 file. prj 的项目文件,其内容如下:</p> <pre>file1. c file2. obj file3. c</pre> <p>在 Project name 项中写入 file. prj,编译时将自动对项目文件中规定的三个源文件分别进行编译,然后连接成 file. exe 文件。如果其中有些文件已经编译成 .OBJ 文件,可直接写上 .OBJ 扩展名,此时将不再编译而只进行连接。当项目文件中的每个文件都无扩展名时,则认为是 C 的源文件。文件名也可以是库文件,但必须写上扩展名 .LIB。</p>
Break make on	中止编译	由用户选择是否有 Warning(警告)、Errors(错误)、Fatal Errors(致命错误)时或 Link(连接)时终止 Make 编译源程序。
Auto dependencies	检查时间	当开关置为 On 时,将检查所有项目文件中原有 .OBJ 目标文件的时间,若该时间与要编译的 .C 源程序不一致,即使源程序的时间较早也要重新编译。当开关置为 off 时,无上述检查。
Clear project	清除项目 文件	清除 Project/Project name 中的项目文件名。
Remove messages	删除信息 窗口内容	把信息窗口中的错误信息清除掉。

(6)Options(选项)菜单

Options 菜单可以设置集成开发环境的工作方式,该菜单的内容及意义见表 C. 6。

表 C. 6

子菜单	说 明
Compiler 编译器	<p>本项选择又有许多二级子菜单,用户可以选择硬件配置、存储模型、调试技术、代码优化、对话信息控制和宏定义等。这些二级子菜单如下:</p> <p>①Model 有 Tiny(微模式)、Small(小模式)、Compact(紧凑模式)、Medium(中模式)、Large(大模式)、Huge(巨模式)六种不同的编译模式供用户选择。</p> <p>②Define 打开一个宏定义框,输入宏定义并按 ENTER 键,即可将其传递给预处理器。多个宏定义之间用“;”分隔,赋值用“=”,前置和后缀空格都被去掉,只保留中间一个空格;若在一个宏中包含分号,必须在分号前用一个反斜线“\”。</p> <p>③Code generation 它又有许多任选项,这些任选项告诉编译器产生什么样的目标代码: Calling convention 可选择 C 或 Pascal 方式传递参数。 Instruction set 可选择 8088/8086 或 80186/80286 指令系列。 Floating point 可选择仿真浮点、数学协处理器浮点或无浮点运算。 Default char type 规定 char 的类型。 Alignment 规定地址对准原则。 Merge duplicate strings 作优化用,将重复的字符串合并在一起。 Standard stack frame 产生一个标准的栈结构。 Test stack overflow 检测堆栈溢出。 Line number 在 .OBJ 文件中放进行号以供调试时用。 OBJ debug information 在 .OBJ 文件中产生调试信息。</p> <p>④Optimization 优化选项。 Optimize for 选择是对程序小型化还是对程序速度进行优化处理。 Use register variable 用来选择是否允许使用寄存器变量。 Register optimization 尽可能使用寄存器变量以减少过多的取数操作。 Jump optimization 通过去除多余的跳转和调整循环与开关语句的办法压缩代码。</p> <p>⑤Source Identifier length 说明标识符有效字符的个数,默认为 32 个。 Nested comments 是否允许嵌套注释。 ANSI keywords only 是只允许 ANSI 关键字还是也允许 Turbo C 2.0 关键字。</p> <p>⑥Error Error stop after 多少个错误时停止编译,默认为 25 个。 Warning stop after 多少个警告错误时停止编译,默认为 100 个。 Display warning 是否显示警告信息。 Portability warning 移植性警告错误。 ANSI violations 与 ANSI 关键字冲突的错误警告。 Common error 常见的错误警告。 Less common error 罕见的错误警告。</p> <p>⑦Names 用于改变段(segment)、组(group)和类(class)的名字,默认值为 CODE、DATA、BSS。一般不要修改此选择项。</p>
Linker 连接器	<p>本菜单设置有关连接的选择项,它有以下内容:</p> <p>①Map file menu 选择是否产生 .MAP 文件。 ②Initialize segments 是否在连接时初始化没有初始化的段。 ③Default libraries 是否在连接其他编译程序产生的目标文件时去寻找其缺省库。 ④Graphics library 是否连接 graphics 库中的函数。 ⑤Warn duplicate symbols 当有重复符号时产生警告信息。 ⑥Stack warning 是否让连接程序产生 No stack 的警告信息。 ⑦Case-sensitive link 连接时是否区分大、小写字母。</p>

(续表)

子菜单	说 明
Environment 环境	<p>本菜单设定集成开发环境,规定是否对某些文件自动存盘及制表键和屏幕大小的设置:</p> <p>①Message tracking Current file 跟踪在编辑窗口中的文件错误。 All files 跟踪所有文件错误。 Off 不跟踪。</p> <p>②Keep message 编译前是否清除 Message 窗口中的信息。</p> <p>③Config auto save 选 on 时,在 Run、Shell 或退出集成开发环境之前,如果 Turbo C 2.0 的配置被改过,则所做的改动将存入配置文件中。选 off 时不保存。</p> <p>④Edit auto save 是否在 Run 或 Shell 之前,自动存储编辑的源文件。</p> <p>⑤Backup file 是否在源文件存盘时产生后备文件(.BAK 文件)。</p> <p>⑥Tab size 设置制表键大小,默认为 8。</p> <p>⑦Zoomed windows 将现行活动窗口放大到整个屏幕,其热键为 F5。</p> <p>⑧Screen size 设置屏幕窗口大小。</p>
Directories 路径	<p>规定编译、连接所需文件的路径,有下列各项:</p> <p>①Include directories 包含文件的路径,多个子目录用“;”分开。</p> <p>②Library directories 库文件路径,多个子目录用“;”分开。</p> <p>③Output directoried 输出文件(.OBJ、.EXE、.MAP 文件)的目录。</p> <p>④Turbo C directoried Turbo C 所在的目录。</p> <p>⑤Pick file name 设置装载的 pick 文件名,如果不设置则从当前编辑过的文件中选取。</p>
Arguments 命令行参数	设定命令行参数,在集成环境中执行程序时,以此项内容作为命令行参数。
Save options 存储配置	保存所有选择的编译、连接、调试和项目到配置文件中,缺省的配置文件为 TCCONFIG.TC。
Retrive options	装入一个配置文件到 TC 中,集成开发环境将使用该配置文件的选择项。

(7)Debug(调试)菜单

该菜单的内容及意义见表 C.7。

表 C.7

子菜单	意义	说 明
Evaluate	赋值	用于调试程序。程序运行时可以改变变量的值。其热键为 Ctrl+F4。有以下三项内容： Expression 表达式 Result 表达式的计算结果 New value 赋给新值
Call stack	堆栈情况	用于调试程序。程序运行时用户可以查看函数调用的情况。其热键为 Ctrl+F3。
Find function	查找函数	在运行时用于显示规定的函数。
Refresh display	刷新显示	如果编辑窗口偶然被用户窗口重写,可用此恢复编辑窗口的内容。

(8) Break/Watch(断点及监视表达式)

该菜单的内容及意义见表 C.8。

表 C.8

子菜单	说 明
Add watch	向监视窗口插入一监视表达式。
Delete watch	从监视窗口中删除当前的监视表达式。
Edit watch	在监视窗口中编辑一个监视表达式。
Remove all watches	从监视窗口中删除所有的监视表达式。
Toggle breakpoint	对光标所在的行设置或清除断点。
Clear all breakpoints	清除源程序中的所有断点。
View next breakpoint	将光标移动到下一个断点处。

9. Turbo C 2.0 的热键

Turbo C 2.0 的热键见表 C.9。

表 C.9

热键名称	功 能
F1	显示当前位置的帮助信息。
F2	把当前编辑的程序存盘后,返回编辑工作。
F3	装入一个文件到编辑窗口。
F4	程序运行到光标所在行。
F5	最大化/还原活动窗口。
F6	切换活动窗口。
F7	调试模式下单步运行程序,跟踪进入函数内部。
F8	调试模式下单步运行程序,不进入函数内部。
F9	编译、连接后生成可执行文件。
F10	切换菜单与活动窗口。
CTRL+F1	调用当前光标处与库函数相关的帮助信息。
CTRL+F2	运行程序复位。
CTRL+F3	观察函数调用栈。
CTRL+F4	调试模式下显示并改变表达式的值。
CTRL+F7	增加一个监视表达式。
CTRL+F8	设置断点 ON 或 OFF。
CTRL+F9	编译、连接后生成可执行文件,并且执行程序。
SHIFT+F10	显示版本信息。
ALT+F1	调出阅读了的最后一个帮助内容。
ALT+F3	选取文件并装入到编辑窗口。
ALT+F5	切换主屏幕和用户屏幕(显示执行程序的结果)。
ALT+F6	切换活动窗口。
ALT+F7	位在上一个错误。
ALT+F8	位在下一个错误。
ALT+F9	编译源程序生成目标程序。
ALT+B	进入 Break/Watch(断点及监视表达式)菜单。
ALT+C	进入 Compile(编译)菜单。
ALT+D	进入 Debug(调试)菜单。
ALT+E	进入 Edit(编辑)窗口。
ALT+F	进入 File(文件)菜单。
ALT+O	进入 Options(选项)菜单。
ALT+P	进入 Project(工程项目)菜单。
ALT+R	进入 Run(运行)菜单。
ALT+X	退出集成开发环境。

10. Turbo C 2.0 的配置文件

所谓配置文件是包含 Turbo C 2.0 有关信息的文件,其中存有编译、连接的选择和路径等信息。

可以用下述方法建立 Turbo C 2.0 的配置:

(1) 建立用户自命名的配置文件

可以从 Options 菜单中选择 Options/Save options 命令,将当前集成开发环境的所有配置存入一个由用户命名的配置文件中。下次启动 TC 时只要在 DOS 下键入:

```
tc /c <用户命名的配置文件名>
```

就会按这个配置文件中的内容作为 Turbo C 2.0 的配置。

(2) TCCONFIG. TC 文件是 Turbo C 2.0 系统默认的配置。也可以执行菜单栏中的

Options/Save options 命令,将当前集成开发环境的所有配置存入 TCCONFIG.TC 配置文件中。若设置 Options/Environment/Config auto save 为 on,则退出集成开发环境时,当前的设置会自动存放到 Turbo C 2.0 配置文件 TCCONFIG.TC 中。Turbo C 在启动时会自动以 TCCONFIG.TC 作为配置文件。

(3)用 TCINST 设置 Turbo C 的有关配置,并将结果存入 TC.EXE 中。Turbo C 在启动时,若没有找到配置文件,则取 TC.EXE 中的默认值。

11. 使用 TCC.EXE 执行命令行编译与连接

在 DOS 状态下,可以键入命令对 C 的源程序进行编译,命令行的一般格式为:

TCC [选项 1 选项 2 ... 选项 n] 源程序文件名 1 源程序文件名 2 ...

其中[选项]是编译程序或连接程序的选择项,各选择项之间用空格隔开。例如:

```
TCC -mt -lt a.c
```

上述命令行把 C 的源程序文件 a.c 编译并连接成 a.com 文件。编译选项-mt 表示用微模式(Tiny Model)编译。连接选项-lt 表示连接生成 COM 文件,而不是 EXE 文件。

在 DOS 命令行上键入:TCC 回车,将显示所有的编译选项。

在 DOS 命令行上键入:TLINK 回车,将显示所有的连接选项,例如显示:

```
/t = create COM file
```

表示在使用 TCC 命令行编译器时,使用-lt 选项连接生成 COM 文件。这里增加一个字母“l”表示 l 后可跟连接(link)选项之意。

附录 D ASCII 字符集

表 D.1 标准 ASCII 字符集

ASCII 值	字符	控制字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符
000	null	NULL	032	space	064	@	096	`
001	☺	SOH	033	!	065	A	097	a
002	●	STX	034	"	066	B	098	b
003	♥	ETX	035	#	067	C	099	c
004	◆	EOT	036	\$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	beep	BEL	039	'	071	G	103	g
008	back space	BS	040	(072	H	104	h
009	■	HT	041)	073	I	105	i
010	line feed	LF	042	*	074	J	106	j
011	♂	VT	043	+	075	K	107	k
012	♀	FF	044	,	076	L	108	l
013	carriage return	CR	045	-	077	M	109	m
014	♪	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	▶	DLE	048	0	080	P	112	p
017	◀	DC1	049	1	081	Q	113	q
018	↑	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	¶	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	■	SYN	054	6	086	V	118	v
023	‡	ETB	055	7	087	W	119	w
024	↑	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[123	{
028	└	FS	060	<	092	\	124	
029	↔	GS	061	=	093]	125	}
030	▲	RS	062	>	094	^	126	~
031	▼	US	063	?	095	_	127	del

表 D.2 扩充 ASCII 字符集

ASCII 值	字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符
128	Ç	160	á	192	Ł	224	α
129	ù	161	í	193	ł	225	β
130	é	162	ó	194	Ť	226	Γ
131	â	163	ú	195	ţ	227	π
132	ä	164	ñ	196	—	228	Σ
133	à	165	Ñ	197	+	229	ο
134	â	166	ª	198	ƒ	230	μ
135	ç	167	º	199	‡	231	τ
136	ê	168	¿	200	Ł	232	φ
137	ë	169	ƒ	201	Ŧ	233	θ
138	è	170	¬	202	⌌	234	Ω
139	ï	171	½	203	⌌	235	δ
140	î	172	¼	204	‡	236	∞
141	ı	173	ı	205	=	237	¢
142	Ä	174	«	206	≠	238	€
143	Å	175	»	207	±	239	∩
144	É	176	☐	208	⌌	240	≡
145	æ	177	☐	209	≠	241	±
146	Æ	178	■	210	π	242	≥
147	ô	179		211	⌌	243	≤
148	ö	180	†	212	ℓ	244	[
149	ò	181	‡	213	ƒ	245]
150	û	182	‡	214	ƒ	246	÷
151	ù	183	π	215	+	247	≈
152	ÿ	184	ƒ	216	+	248	°
153	Ö	185	‡	217	┘	249	•
154	Ü	186		218	ƒ	250	•
155	ø	187	ƒ	219	■	251	√
156	£	188	┘	220	■	252	°
157	¥	189	⌌	221	■	253	²
158	Pt	190	┘	222	■	254	■
159	f	191	┘	223	■	255	blank

附录 E 运算符的优先级与结合性

运算符的优先级,从上至下优先级降低:

运算符	结合性
()、[]、->、.	从左至右
++、--、+、-、!、~、(类型)、*、&、sizeof	从右至左
*, /、%	从左至右
+, -	从左至右
<<、>>	从左至右
<、<=、>、>=	从左至右
==、!=	从左至右
&	从左至右
^	从左至右
!	从左至右
&&	从左至右
	从左至右
?:	从右至左
=、+=、-=、*=、/=、%=、&=、^=、!=、<<=、>>=	从右至左
,	从左至右

注:运算符第二行的“+、-”号代表“正、负”号;第四行的“+、-”号代表“加、减”号。
运算符第二行的“*”号为指针运算符;第三行的“*”号为算术运算符“乘”号。

附录 F Turbo C 的部分标准函数

下面列表给出了 Turbo C 的部分库函数的功能索引,并没有列出各函数的所有细节。

1. 输入输出及相关函数 (stdio. h)

函数原型	功能简要说明
FILE * fopen(char * fname, char * mode);	打开指定文件
FILE * freopen(char * fname, char * mode, FILE * fp);	首先关闭指定流,然后再打开文件
int fclose(FILE * fp);	关闭指定流,释放文件缓冲区
int fflush(FILE * fp);	刷新缓冲区
int remove(char * fname);	删除指定文件
int rename(char * oldname, char * newname);	更改文件名
FILE * tmpfile(void);	打开一个临时文件
char * tmpnam(char * name);	产生一个独特的文件名字
int setvbuf(FILE * fp, char * buf, int mode, unsigned size);	设置 fp 的缓冲区及模式与大小
int fprintf(FILE * fp, char * format, ...);	格式输出到 fp 指定文件
int printf(char * format, ...);	格式输出到标准输出设备
int sprintf(char * buf, char * format, ...);	格式输出到字符数组 buf
int fscanf(FILE * fp, char * format, ...);	从指定文件格式读入数据
int scanf(char * format, ...);	从标准输入设备格式读入数据
int sscanf(char * buf, char * format, ...);	从字符数组 buf 中格式读入数据
int fgetc(FILE * fp);	从 fp 指定文件读入一个字符
int fgets(FILE * fp);	从 fp 指定文件读入一个字符串
int fputc(int ch, FILE * fp);	输出字符 ch 到指定文件中
int fputs(char * str, FILE * fp);	输出字符串 str 到指定文件中
int getc(FILE * fp);	从 fp 指定文件读入一个字符
int getchar(void);	从标准输入设备读入一个字符
char * gets(char * s);	从标准输入设备读入一个字符串
int putc(int ch, FILE * fp);	输出字符 ch 到 fp 指定文件中
int putchar(int ch);	输出字符 ch 到标准输出设备中
int puts(char * str);	输出字符串 str 到标准输出设备中
int fread(void * buf, int size, int count, FILE * fp);	从 fp 指定文件读入若干数据块
int fwrite(void * buf, int size, int count, FILE * fp);	写若干数据块到 fp 指定文件中
int fseek(FILE * fp, long offset, int origin);	定位文件位置指针
long ftell(FILE * fp);	报告文件位置指针
void rewind(FILE * fp);	将文件位置指针置于文件开始位置
void clearerr(FILE * fp);	将出错标记置零,重置文件结束标志
int feof(FILE * fp);	检测文件结束标志
int ferror(FILE * fp);	检测文件错误
void perror(char * str);	映射出错信息字符串

2. 字符处理函数 (ctype.h)

函数原型	功能简要说明
int isalnum(int ch);	判断 ch 是否为字母或数字
int isalpha(int ch);	判断 ch 是否为字母
int isascii(int ch);	判断 ch 是否为标准 ASCII 字符
int iscntrl(int ch);	判断 ch 是否为控制字符
int isdigit(int ch);	判断 ch 是否为数字
int isxdigit(int ch);	判断 ch 是否为十六进制的数字
int isgraph(int ch);	判断 ch 是否为除空格以外的可打印字符
int isprint(int ch);	判断 ch 是否为可打印字符
int ispunct(int ch);	判断 ch 是否为标点符号
int isspace(int ch);	判断 ch 是否为空格、制表符或换行符
int islower(int ch);	判断 ch 是否为小写字母
int isupper(int ch);	判断 ch 是否为大写字母
int tolower(int ch);	将大写字母转换为小写字母
int toupper(int ch);	将小写字母转换为大写字母

3. 字符串处理函数 (string.h)

函数原型	功能简要说明
char * strcpy(char * str1, char * str2);	将字符串 str2 复制到字符串 str1 中
char * strncpy(char * str1, char * str2, int n);	将字符串 str2 前 n 个字符复制到字符串 str1 中
char * strcat(char * str1, char * str2);	将字符串 str2 复制到字符串 str1 后
char * strncat(char * str1, char * str2, int n);	将字符串 str2 前 n 个字符复制到字符串 str1 后
char * strcmp(char * str1, char * str2);	比较两个字符串 str1 和 str2 的大小
char * strncmp(char * str1, char * str2, int n);	比较两个字符串 str1 和 str2 的前 n 个字符大小
char * strchr(char * str, int ch);	寻找字符 ch 在字符串 str 中首次出现的位置指针
char * strrchr(char * str, int ch);	寻找字符 ch 在字符串 str 中最后出现的位置指针
int strspn(char * str1, char * str2);	在 str1 中寻找第一个不属于 str2 的字符位置
int strcspn(char * str1, char * str2);	计算 str1 的初始子串(不含 str2 中的字符)长度
char * strpbrk(char * str1, char * str2);	在 str1 中寻找第一个属于 str2 的字符位置指针
char * strstr(char * str1, char * str2);	寻找 str1 中第一个与 str2 相匹配的子串指针
int strlen(char * str);	计算字符串 str 的长度
char * strerror(int errnum);	报告错误信息
char * strtok(char * str1, char * str2);	将 str2 看成 str1 的子串分隔符, 各次调用依次返回各子字符串
char * strset(char * str, char ch);	将字符串 str 中所有字符都设置为 ch 值
char * strnset(char * str, char ch, unsigned n);	将字符串 str 的前 n 个字符设置为 ch 值
char * strrev(char * str);	把字符串 str 中的所有字符的顺序都颠倒过来
char * strlwr(char * str);	把字符串 str 中的所有字母都变为小写字母
char *strupr(char * str);	把字符串 str 中的所有字母都变为大写字母

4. 数学函数 (math.h)

函数原型	功能简要说明
double sin(double x);	计算 x 的正弦值
double cos(double x);	计算 x 的余弦值
double tan(double x);	计算 x 的正切值
double asin(double x);	计算 x 的反正弦值
double acos(double x);	计算 x 的反余弦值
double atan(double x);	计算 x 的反正切值
double atan2(double x, double y);	计算 x/y 的反正切值
double sinh(double x);	计算 x 的双曲正弦值
double cosh(double x);	计算 x 的双曲余弦值
double tanh(double x);	计算 x 的双曲正切值
double exp(double x);	计算以自然数 e 为底 x 为幂的指数值
double pow(double x, double y);	计算 x 为底 y 为幂的指数值
double log(double x);	计算 x 的自然对数值
double log10(double x);	计算以 10 为底的 x 的对数值
double sqrt(double x);	计算 x 的平方根值
double ceil(double x);	计算不小于 x 的最小整数值(表示为双精度)
double floor(double x);	计算不大于 x 的最大整数值(表示为双精度)
double ldexp(double x, int exp);	计算 x 与 2 的 exp 次幂的乘积值
double frexp(double y, int *exp);	计算方程 $y = ldexp(x, exp)$ 成立时的 x 值
double modf(double x, int *i);	计算 x 的小数部分值, 整数部分放在 i 中
double fmod(double x, double y);	计算 x/y 的余数值
double fabs(double x);	计算 x 的绝对值(双精度)
int abs(int j);	计算 j 的绝对值(整型)
long labs(long j);	计算 j 的绝对值(长整型)
double cabs(struct complex z);	计算复数 z 的绝对值
double poly(double x, int n, double c[]);	计算 x 的 n 次多项式值, c 为系数数组
double hypot(double x, double y);	计算以 x 和 y 为直角边的直角三角形的斜边长度

5. 数据转换函数 (stdlib.h)

函数原型	功能简要说明
double atof(char *str);	将 str 转换为一个双精度实数
int atoi(char *str);	将 str 转换为一个整数
long atoll(char *str);	将 str 转换为一个长整型数
double strtod(char *str, char **end);	将 str 转换为一个双精度实数
long strtol(char *str, char **end, int radix);	将 str 转换为一个 radix 进制的(无符号)长整型数
unsigned long strtoul(char *str, char **end, int radix);	将 str 转换为一个 radix 进制的(有符号)长整型数
char * itoa(int num, char *str, int radix);	把 num 转换为 radix 进制表示的字符串
char * fcvt(double value, int ndigit, int *dec, int *sign);	将 value 转换成长度为 ndigit 的字符串, *dec 为小数点位置, *sign 记录正负号
char * ecvt(double value, int ndigit, int *dec, int *sign);	将 value 转换成长度为 ndigit 的字符串
char * gcvt(double value, int ndigit, char *buf);	将 value 转换成长度为 ndigit 的字符串

6. 存储分配函数 (alloc. h)

函数原型	功能简要说明
<code>int * calloc(unsigned num, unsigned size);</code>	申请 <code>num * size</code> 个字节的内存块
<code>int * malloc(unsigned size);</code>	申请 <code>size</code> 个字节的内存块
<code>void * realloc(void * ptr, unsigned newsize);</code>	重新分配 <code>newsize</code> 个字节的内存块给 <code>ptr</code>
<code>void free(void * ptr);</code>	释放 <code>ptr</code> 所指向的内存块
<code>unsigned [long] coreleft(void);</code>	计算堆中剩余(未使用)的内存字节数

7. 其他函数

函数原型	功能简要说明	头文件
<code>int rand(void);</code>	产生 0 到 <code>RAND_MAX</code> 之间的随机数	<code>stdlib. h</code>
<code>int random(int num);</code>	产生 0 到 <code>num</code> 之间的随机数	<code>stdlib. h</code>
<code>void randomize(void);</code>	初始化随机数发生器	<code>stdlib. h</code>
<code>void srand(unsigned seed);</code>	建立 <code>rand()</code> 产生伪随机数的起始点	<code>stdlib. h</code>
<code>void exit(int status);</code>	使程序运行正常终止	<code>process. h</code>
<code>void abort(void);</code>	使程序运行非正常终止	<code>process. h</code>
<code>int exec...()</code> 系列函数	用于执行另外一个程序	<code>process. h</code>
<code>int system(char * command);</code>	把 <code>command</code> 传给系统并当成命令来执行	<code>process. h</code>
<code>int chdir(char * path);</code>	将 <code>path</code> 所指的目录变成当前目录	<code>dir. h</code>
<code>int rmdir(char * path);</code>	删除 <code>path</code> 所指的目录	<code>dir. h</code>
<code>int getcurdir(int drive, char * dir);</code>	获得 <code>drive</code> 所指的驱动器的当前目录	<code>dir. h</code>
<code>char * getcwd(char * char, int len);</code>	获得当前目录的全路径名(最多 <code>len</code> 个字符)	<code>dir. h</code>
<code>char * serachpath(char * fname);</code>	用 <code>DOS PATH</code> 环境变量找出 <code>fname</code> 所指文件	<code>dir. h</code>
<code>char * mktemp(char * fname);</code>	产生一个惟一的文件名	<code>dir. h</code>
<code>int getdisk(void);</code>	返回当前驱动器的代码	<code>dir. h</code>
<code>int setdisk(int drive);</code>	设置 <code>drive</code> 所指驱动器为当前驱动器	<code>dir. h</code>
<code>void getdate(struct date * d);</code>	获得当前日期信息	<code>time. h</code>
<code>void gettime(struct time * t);</code>	获得当前时间信息	<code>time. h</code>
<code>void setdate(struct date * d);</code>	设置当前日期	<code>time. h</code>
<code>void settime(struct time * t);</code>	设置当前时间	<code>time. h</code>

附录 G 编译错误信息

Turbo C 编译系统在编译源程序时会产生三种类型的错误：致命错误、一般错误和警告。其中致命错误一般是内部编译出错。当一个致命错误出现时，编译立即停止，必须采取适当的措施并重新启动编译系统才能重新使用。一般错误是指程序的语法错误、磁盘或内存存取错误或命令行错误等。编译程序遇到这类错误时，将继续完成现阶段的编译。编译系统在每一个阶段（编译预处理、语法分析、优化、代码生成等）尽可能多地找出源程序中的错误，以便程序设计人员修改程序中的错误。警告则只是指出一些值得怀疑的情况，它并不阻止编译及连接的进行。

出现错误时编译系统首先输出这三类信息，然后输出源文件名及编译程序发现出错处的行号。这里应该注意：Turbo C 并限定在某一行设置语句，因此，真正产生错误的原因就可能出现在所提示行号的前面一行或几行。最后输出错误信息的内容。

下面按字母顺序分别列出致命错误和一般错误的信息。

1. 致命错误

(1) Bad call of in-line function (非法调用内部函数)

在使用一个宏定义的内部函数时没有正确地调用。一个内部函数是以两个下划线(__)开始和结束的。

(2) Irreducible expression tree (不可约表达式树)

这种错误指的是文件行中的表达式不合法，使得代码生成程序无法为它生成代码。这种表达式必须避免使用。

(3) Register allocation failure (存储器分配失败)

这种错误指的是文件行中的表达式太复杂，代码生成程序无法为它生成有效代码。此时应简化这种繁杂的表达式或干脆避免使用它。

2. 一般错误

(1) # operator not followed by macro argument name (# 运算符后没跟宏变元名)

在宏定义中，# 后必须跟一个标识符(宏名)，否则出错。

(2) "xxxx" not an argument ("xxxx" 不是函数参数)

在源程序中将该标识符定义为一个函数参数，但是此标识符没有在函数参数表中出现。

(3) Ambiguous symbol "xxxx" (二义性符号 "xxxx")

两个或多个结构可能存在某一相同的域名(结构分量)，它们属于不同的变量，故其所具有的偏移、类型是可以不同的。如果在变量或表达式中引用这些结构分量而未带结构名时会产生二义性。此时需修改某个域名或在引用时加上结构名。

(4) Argument # missing name (参数 # 名丢失)

参数名已脱离用于定义函数的函数原型。如果函数以原型定义，该原型必须包含所有的参数名。

(5) Argument list syntax error (参数表出现语法错误)

函数调用的参数间必须以逗号隔开，并以一个右括号结束。若源文件中含有一个其后既不

是逗号又不是右括号的参数则出现此类错误。

(6)Array bounds missing "]"(数组的界限符"]"丢失)

在源文件中定义了一个数组,但此数组没有以一个右方括号结束。

(7)Array size too large(数组长度太大)

定义的数组太大,超过了可用的内存空间。

(8)Assembler statement too long(汇编语句太长)

C 语言规定,在 C 的源程序中直接插入的汇编语句最长不能超过 480 字节。

(9)Bad configuration file(配置文件不正确)

TURBOC.CFG 配置文件中包含的不是合适命令行选择项的非注释文字。配置文件选择项必须以一个短横线开始。

(10)Bad file name format in include directive(包含指令中文件名格式不正确)

包含文件名必须用引号("filename.h")或尖括号(<filename.h>)括起来,否则将产生本类错误。如果使用了宏,则产生的扩展文本是不正确的,因为使用宏常常出现无引号的现象,所以无法识别。

(11)Bad ifdef directive syntax(ifdef 指令语法错误)

#ifdef 必须以单个标识符(只此一个)作为该指令的体。

(12)Bad ifndef directive syntax(ifndef 指令语法错误)

#ifndef 必须以单个标识符(只此一个)作为该指令的体。

(13)Bad undef directive syntax(undef 指令语法错误)

#undef 必须以单个标识符(只此一个)作为该指令的体。

(14)Bad file size syntax(位字段长语法错误)

一个位字段长必须是 1~16 位的常量表达式。

(15)Call of non-function(调用未定义的函数)

程序调用一个函数,但此函数在此程序中没有定义。这种错误通常是由于不正确的函数说明或函数名拼写错误而造成的。

(16)Can not modify a const object(不能修改一个常量对象)

对定义为常量的对象进行的不合法操作(如常量赋值)引起本错误。

(17)Case outside of switch(case 语句出现在 switch 外面)

编译程序发现 case 语句出现在 switch 语句的外面。这类故障通常是由于括号不匹配造成的。

(18)Case statement missing(漏掉 case 语句)

case 语句必须包含一个以冒号结束的常量表达式。如果漏掉了冒号或在冒号前加上了其他字符,就会出现这类错误。

(19)Character constant too long(字符常量太长)

字符常量的长度只能是一个或两个字长,如果超过此长度则会出现本类错误。

(20)Compound statement "}"(复合语句中漏掉"}")

编译程序扫描到文件结束时,未发现复合语句的结束符(即"}"),此类错误通常是由于大括号不匹配所引起的。

(21)Conflicting type modifiers(类型修饰符前后矛盾)

对同一指针,只能指定一种变址修饰符(如 near 或 far);而对于同一函数,则只能给出一

种语言修饰符(如 Cdecl、pascal 或 mteruprt),如果源程序中类型修饰符前后矛盾,则会出现此类错误。这种错误在使用多种语言混合编程时出现,使用单一的语言编写程序时,由于不用到这类修饰符,所以不会有这种错误出现。

(22)Constant expression required(需要常量表达式)

在需要常量表达式的地方(如定义数组时,下标表达式就应该是常量表达式)使用了变量。这种错误通常是由于宏定义 #define 中常量符号拼写错误引起的。

(23)Could not find file "xxxx"(文件“xxxx”找不到)

编译程序找不到命令行上指定的文件。

(24)Declaration missing(漏掉了必要的说明)

如果源文件中包含了一个 struct 或 union 域声明,而后面却漏掉了分号,则会出现此类错误。

(25)Declaration needs type or storage class(在进行变量说明时没有指明变量的类型和存储类别)

在进行变量定义时必须指明变量的类型,否则,就会出现这类错误。

(26)Declaration syntax error(变量定义时出现语法错误)

在源文件中,如果定义变量的语句中丢失了某些符号或输入了多余的符号,则会出现此类错误。

(27)Default out side of switch(default 语句出现在 switch 语句的外面)

这类错误通常是由于括号不配对而引起的。

(28)Define directive needs an identifier(Define 语句中必须要有一个标识符)

#define 后面的第一个非空的字符串必须是一个合法的 C 语言标识符。若在该位置上出现一个不是标识符的字串,则会出现这类错误。

(29)Division by zero(在进行数学运算时,有数被零除)

当源程序中的常量表达式出现除数为零的情况时,会造成这类错误。

(30)Do statement must have while(do 循环语句中没有 while)

C 语言中,do~while 语句必须配对使用。如果源程序中出现了一个没有关键字 while 的 do 语句,则会出现这类错误。

(31)Do while statement mission (" (do~while 语句中漏掉了“(”)

(32)Do while statement mission ") " (do~while 语句中漏掉了“)”)

C 语言的 do~while 语句中,关键字 while 后面必须接一对圆括号()。如果在 do 语句中,关键字 while 后面缺少了括号,则会出现此类错误。

(33)Do while statement missing "; " (do~while 语句中漏掉了“;”)

C 语言规定:所有的语句都必须以“;”作为结束标志。如果在 do 语句的条件表达式中右括号后面漏掉了分号“;”,则会出现这类错误。

(34)Duplicate case(在 switch 语句中,case 的情况值不惟一)

switch 语句中的每个 case 后必须有一个惟一的常量表达式的值,否则会出现这类错误。

(35)Enum syntax error(枚举类型 enum 中,出现语法错误)

若 enum 说明的标识符表达式有错误,则会导致此类错误的发生。

(36)Enumeration constant syntax error(枚举常量语法错误)

若赋给 enum 类型变量的表达式的值不为常量,则会出现此类错误。

(37)Error directive:XXXX (Error 指令:XXXX)

C 语言的编译系统在处理源文件中的“#error”指令时,显示该指令指出的信息。

(38)Error writing output file(在输出文件上写数据时出现错误)

这类错误通常是由于所用磁盘已满或处于写保护状态,因而无法进行写操作而造成的。出现这类错误时,应更换一块工作磁盘或删除工作盘上不必要的文件。

(39)Expression syntax(表达式语法错误)

此类错误通常是由于出现了两个连续的操作符,括号不配对或缺少括号,前一句漏掉了分号等原因而引起的。

(40)Extra parameter in call(函数调用时出现多余参数)

一般情况下,进行函数调用时实参个数应与形参个数相等。如果在调用一个函数时实际参数个数多于函数形式参数的个数,则会出现这类错误。

(41)Extra parameter in call to XXXX(调用函数 XXXX 时出现了多余参数)

(42)File name too long(文件名太长)

#include 给出的文件名太长时,编译系统将无法处理,因此出错。通常 DOS 系统下的文件名(含路径)长度不能超过 64 个字符。

(43)For statement missing "(" (for 语句中缺少“()”)

(44)For statement missing ")" (for 语句中缺少“)”)

在 for 语句中,如果控制表达式后缺少括号,则会出现这类错误。

(45)For statement missing ";" (for 语句中缺少“;”)

在 for 语句中,如果某个表达式后缺少逗号,则会出现这类错误。

(46)Function Call missing ")" (函数调用时缺少“)”)

C 语言中进行函数调用时,括号必须配对使用。如果调用函数时,在实参表列后漏掉了“)”或括号不匹配,则会出现这类错误。

(47)Function definition out of place(定义函数时位置不正确)

C 语言规定:函数不能嵌套定义,即在一个函数内部不能定义另外的函数。如果试图在一个函数内部定义另一个函数,则会出现此类错误。

(48)Function doesn't take a variable number of argument(函数不接受可变的参数个数)

如果在源程序中的某个函数内使用了 va-start 宏,则此函数不能接受可变数量的参数。

(49)Goto statement missing label(goto 语句后没有指明标号)

关键字 goto 后必须有一个标识符。

(50)If statement missing "(" (if 语句缺少“()”)

(51)If statement missing ")" (if 语句缺少“)”)

(52)Illegal initialization(初始化不合法)

对变量(包括简单变量、数组、结构体数据等)进行初始化必须使用常量表达式,或是一个全局变量 extern 或 static 的地址加减一个常量。

(53)Illegal octal digit(非法八进制数)

在八进制数中不允许出现数字 8 和 9。如果源程序中一个以 0 开头的数(按 C 语言的规定,这是一个八进制数)中出现了数字 8 和 9,或其他非法字符,则会出现这类错误。

(54)Illegal pointer subtraction(非法的指针减法运算)

C 语言规定:非指针变量不能减去一个指针变量。如果在一个 C 的源程序中出现这种减

法,则会发生本错误。

(55)Illegal structure operation(对结构体的操作非法)

结构体只可以使用点(.)、取地址(&)和赋值(=)运算符,或者作为函数的参数进行传递。如果编译程序发现对结构体使用了其他运算符时,则会出现此类错误。

(56)Illegal use of floating point(非法的浮点运算)

浮点运算分量不允许出现在各种位运算符中。如果编译系统发现位操作中使用了浮点实数,则产生出错信息。

(57)Illegal use of pointer(使用指针不合法)

施于指针的运算仅可以是加、减、赋值、比较、取值(*)或箭头(→)。若对指针变量使用其他运算,则会出现这类错误。

(58)Improper use of a typedef symbol(typedef 符号的使用不合法)

(59)Incompatible storage class(存储类别不相容)

在 C 语言源程序的函数内部不能使用关键字 extern,只能使用 static 或 auto(或干脆不指明存储类别)。

(60)Incompatible type conversion(类型转换不相容)

如果源程序试图把一种数据类型转换成另一种数据类型,而这两种数据类型是不可以相互转换的,则会出现这种错误,如函数与非函数之间的转换、浮点数与指针之间的转换。

(61)Incorrect command line argument:XXXX(不正确的命令行参数:XXXX)

(62)Incorrect configuration file argument:XXXX(不正确的配置文件参数:XXXX)

(63)Incorrect number format(数据格式不正确)

编译程序发现在十六进制数据中出现十进制小数点。

(64)Incorrect use of default(default 语句使用不合法)

如果编译程序发现在 default 关键字后缺少冒号,则出现这类错误。

(65)Initializer syntax error(对数据进行初始化时有错误)

(66)Invalid indirection(无效的取值运算)

取值运算符(*)要求非空的指针变量作为运算分量。

(67)Invalid macro argument separator(宏参数分隔符非法)

(68)Invalid pointer addition(指针相加运算时有错)

指针与指针不能相加。如果源程序中试图用一个指针与另一个指针相加,则会出现此类错误。

(69)Invalid use of arrow(非法使用箭头运算符→)

(70)Invalid use of dot(点运算符使用错误)

点运算符(.)用于指向结构体变量的成员,其后必须紧跟一个标识符(结构体成员名),否则出现这类错误。

(71)A value required(要求赋值)

(72)Macro argument syntax error(宏参数语法错误)

(73)Macro expansion too long(宏扩展太长)

一个宏不能扩展超过 4096 个字符。当宏递归扩展自己时常出现此类错误。

(74)Mismatch number of parameters in definition(在函数定义中参数个数不匹配)

函数定义中的参数和函数原型中提供的信息不匹配。

(75) Misplaced break (break 语句的位置不正确)

编译程序发现 break 语句在 switch 语句或循环结构的外面。

(76) Misplaced continue (continue 语句的位置不正确)

编译程序发现 continue 语句在 switch 语句或循环结构的外面。

(77) Misplaced decimal point (十进制小数点的位置不正确)

编译程序发现浮点常数的指数部分有一个小数点。

(78) Misplaced else (else 语句的位置不正确)

else 语句必须与 if 语句配对连用, 否则就会产生这类错误。此类错误的产生, 除了由于 else 多余以外, 还可能由于多余的分号、漏写了花括号、或者前面的 if 语句错误引起的。

(79) Misplaced endif directive (endif 指令的位置不正确)

编译程序找不到与 #endif 指令匹配的 #if、#ifdef 或 #ifndef 指令。

(80) Must be addressable (必须是可编址的)

取地址运算符 & 必须作用于一个可编址的对象。如果将 & 用于一个不可编址的对象如寄存器变量, 则会出现此类错误。

(81) Must take address of memory location (必须是内存中的一个地址)

如果源文件将取地址运算符 & 用于不可编址的表达式, 则会出现这类错误。

(82) No file name ending (没有文件名结束符)

在 #include 命令中, 文件名缺少正确的闭引号 (") 或尖括号 (>)。

(83) No file names giving (没有给出文件名)

Turbo C 的编译命令 (TCC) 中没有包含文件名。

(84) Non-portable pointer assignment (对不可移植的指针赋值)

Turbo C 不允许将一个地址值赋给一个非指针变量及将非地址值赋给指针变量。如果程序进行了这种赋值, 则会引起此错误。作为一个特例, 把常量零赋给一个指针变量是允许的。

(85) Non-portable pointer comparison (对不可移植的指针进行比较)

一个指针与一个非指针变量是不允许进行比较的。如果源程序中将一个指针变量与一个非指针变量进行比较, 则会产生此类错误。

(86) Non-portable return type conversion (不可移植的返回类型转换)

函数的返回语句中的表达式类型通常应与函数说明中规定的函数值类型相同。如果它们的类型不同, 则会将表达式的类型转换成函数值类型, 但这种转换必须是可以进行的, 否则就会出现这类错误。

(87) Not an allowed type (使用了不允许的类型)

该错误指出源程序中使用了禁止使用的数据类型, 如 return 语句返回一个函数或一个数组等。

(88) Out of memory (内存不够)

(89) Pointer required on left side of —> (在运算符 —> 的左侧必须使用指针)

(90) Redclaration of "XXXX" (对 "XXXX" 进行重定义)

在一个函数中, 一个标识符只能定义一次。如果源程序在定义了一个标识符后又对它进行定义, 则会出现这种错误。

(91) Size of structure or array not known (结构体或数组的大小不定)

有些表达式 (如 sizeof 或存储说明) 中不允许出现没有定义的结构体或数组。如果在这些

表达式中误用了未定义的结构体或数组,则会引起本错误。

(92)Statement missing ";"(语句缺少";")

(93)Structure or union syntax error(结构体或联合体语法错误)

编译程序发现在关键字 struct 或 union 后面没有标识符或大括号"{"。

(94)Structure size too long(结构太长)

(95)Subscripting missing "]"(数组的下标缺少"]")

(96)Switch statement missing "("(switch 语句缺少"(")

(97)Switch statement missing ")"(switch 语句缺少")")

(98)Too few parameter in call(进行函数调用时参数太少)

(99)Too few parameter in call to "XXXX"(调用函数"XXXX"时参数太少)

(100)Too many cases(case 语句太多)

Turbo C 中,一个 switch 语句最多只能使用 257 个 case。

(101)Too many decimal points(一个十进制数中出现多个小数点)

(102)Too many default in case(case 语句中出现多个 default 语句)

(103)Too many exponents(阶码太多)

如果编译时发现一个浮点常量中不止一个阶码,则会出现这类错误。

(104)Too many initializers(初始化太多)

编译时发现初始化比说明所允许的要多。

(105)Too many storage classes in declaration(说明中存储类别太多)

一个说明语句中只允许有一种存储类别。

(106)Too many types in declaration(说明中类型太多)

一个说明中只允许有一种下列基本类型:char、int、float、double、struct、union、enum 或 typedef 定义的类型名。

(107)Too much auto memory in function(函数中自动型存储变量太多)

所使用的变量超过了内存所允许使用的存储空间。

(108)Too much globe data define in file(文件中定义的全局类型变量太多)

一个文件中的所有全局变量的总数不得超过 64k 字节,否则将引起这类错误。

(109)Two consectutive dots(程序中出现两个连续的点)

(110)Type mismatch in parameter # (参数"#"的类型不匹配)

通过一个指针访问已由原型说明的函数时,所给定实际参数不能转换为已说明的参数类型。

(111)Type mismatch in parameter # in call to "XXXX"(调用函数"XXXX"时参数#的类型不匹配)

(112)Type mismatch in parameter "XXXX"(参数"XXXX"的类型不匹配)

(113)Type mismatch in parameter "XXXX" in call _ to "YYYY"(在调用函数"YYYY"时,参数"XXXX"不匹配)

(114)Type mismatch in redeclaration of "XXXX"(重定义"XXXX"时类型不匹配)

(115)Unable to creat output file "XXXX"(不能建立输出文件"XXXX")

(116)Unable to creat turboc. lnk(不能建立临时文件 turboc. lnk)

(117)Unable to execute command "XXXX"(不能执行"XXXX"命令)

(118)Unable to open include file "XXXX"(不能打开包含文件“XXXX”)

(119)Unable to open input file "XXXX"(不能打开输入文件“XXXX”)

(120)Undefined lable "XXXX"(标号“XXXX”没有定义)

函数中出现在 goto 语句后面的标号无定义。

(121)Undefined structure "XXXX"(结构体“XXXX”没有定义)

源文件中某些行使用了某个结构,但此结构未经说明。这可能是由于结构名拼写错误或缺少结构说明而引起的。

(122)Undefined symbol "XXXX"(符号“XXXX”没有定义)

(123)Unexpected end of file in comment started on line #(源文件在#行开始的注释中意外结束)

这种错误通常是由于丢失注解结束标记(* /)而引起的。

(124)Unexpected end of file in conditional started on line #(源文件在#行开始的条件语句中意外结束)

(125)Unknown preprocessor directire "XXXX"(非法的编译预处理命令“XXXX”)

(126)Unterminated character constant(没有结束的字符常量)

(127)Unterminated string(没有结束的字符串)

(128)Unterminated string or character constant(没有结束的字符串或字符常量)

(129>User break(用户中断)

(130)While statement missing "(" (while 语句中漏掉了“(”)

(131)While statement missing ")" (while 语句中漏掉了“)”

(132)Wrong number of arguments in call of "XXXX"(调用函数“XXXX”时参数个数错误)

附录 H 习题参考答案

第 2 章 C 语言的语法基础

一、选择题

- | | | | | |
|--------|--------|--------|--------|--------|
| 2.1 D | 2.2 C | 2.3 A | 2.4 B | 2.5 B |
| 2.6 C | 2.7 D | 2.8 B | 2.9 D | 2.10 B |
| 2.11 A | 2.12 D | 2.13 A | 2.14 A | 2.15 C |
| 2.16 D | 2.17 B | 2.18 C | 2.19 A | 2.20 C |
| 2.21 A | 2.22 A | 2.23 D | 2.24 C | 2.25 C |
| 2.26 B | 2.27 D | 2.28 A | 2.29 B | 2.30 A |
| 2.31 B | 2.32 C | 2.33 B | 2.34 B | 2.35 B |
| 2.36 A | 2.37 A | 2.38 D | | |

二、填空题

- 2.39 【1】 15 或 0xf 或 017(其中 x, f 大小写均正确)
- 2.40 【2】 1,2,0
- 2.41 【3】 123, 45, #, 6.789000, 123.000000
- 2.42 【4】 $x=4.567890, x=4.568, x=4.56789000, x=+5,$
 $x=4.56789, x=4.56789e+00$
- 2.43 【5】 关键字、预定义标识符、用户定义标识符
- 2.44 【6】 十进制、八进制、十六进制
- 2.45 【7】 -128~127 与 0~255

第 3 章 程序控制结构

一、选择题

- | | | | | |
|-------|-------|-------|-------|-------|
| 3.1 A | 3.2 D | 3.3 B | 3.4 C | 3.5 A |
|-------|-------|-------|-------|-------|

二、填空题

- | | | | |
|---------|--------|-----|-------|
| 3.6 【1】 | 50 | 【2】 | i |
| 3.7 【3】 | num%10 | 【4】 | max=t |
| 3.8 【5】 | j%2==1 | | |

第 4 章 构造型数据类型

一、选择题

- | | | | | |
|--------|--------|--------|--------|--------|
| 4.1 D | 4.2 C | 4.3 A | 4.4 B | 4.5 B |
| 4.6 C | 4.7 D | 4.8 D | 4.9 A | 4.10 A |
| 4.11 C | 4.12 D | 4.13 C | 4.14 D | 4.15 B |
| 4.16 A | 4.17 C | 4.18 A | 4.19 A | 4.20 C |
| 4.21 A | 5.22 D | 5.23 B | 4.24 B | 4.25 B |
| 4.26 B | | | | |

二、填空题

- 4.27 【1】 $w[i-1]$ 【2】 $w[p]$
 4.28 【3】 15,9,6,5,1,(插入法排序)
 4.29 【4】 $a[k]! = y$
 4.30 【5】 $s[\text{len}]$ 或 $s[\text{len}]! = '\backslash 0'$ 或 $s[\text{len}]! = 0$
 4.31 【6】 4,8
 4.32 【7】 $\text{tab}[i][0]$ 【8】 $j++$ 或 $j+=1$ 或 $j=j+1$
 4.33 【9】 char 【10】 $\text{daytab}[m-1]$ 【11】 7
 4.34 【12】 $a[i][j]! = a[j][i]$ 或 $a[j][i]! = a[i][j]$ 【13】 1 或非零值
 4.35 【14】 DDBCC

第5章 指针

一、选择题

- 5.1 D 5.2 D 5.3 D 5.4 A 5.5 C
 5.6 C 5.7 B 5.8 C 5.9 D 5.10 D
 5.11 D 5.12 B 5.13 B 5.14 D 5.15 B
 5.16 A 5.17 A 5.18 A 5.19 D 5.20 C
 5.21 D 5.22 C 5.23 A 5.24 C 5.25 B
 5.26 D 5.27 D 5.28 A 5.29 D 5.30 B

二、填空题

- 5.31 【1】 i 【2】 $*pe$ 【3】 $pe--$
 5.32 【4】 $m+n-1$ 【5】 $*sub='\backslash 0'$ 或 $*sub=0$
 5.33 【6】 $*p$ 或 $p[0]$
 5.34 【7】 score 【8】 $[n-1][i]$
 5.35 【9】 $n++$ 或使 n 值加 1 的表达式 【10】 $*p>'9'$
 【11】 $'0'$ 或 48
 5.36 【12】 $*p==*s$ 或 $*s==*p$ 【13】 $s++$

第6章 函数

一、选择题

- 6.1 D 6.2 B 6.3 B 6.4 B 6.5 D

二、填空题

- 6.6 【1】 void sub 【2】 sub 【3】 int m 【4】 $m\%10$ 【5】 $a1==5 \parallel a2==5$

第7章 数组、指针、函数的应用

一、选择题

- 7.1 A 7.2 B 7.3 A 7.4 D 7.5 C
 7.6 B 7.7 D 7.8 C 7.9 D 7.10 C
 7.11 B 7.12 A 7.13 C 7.14 D

二、填空题

- 7.15 【1】 $(*a)[N]$

- 7.16 【2】 *s
 7.17 【3】 ehco My computer
 7.18 【4】 *p->next

第 8 章 文件

一、选择题

- 8.1 D 8.2 C 8.3 B 8.4 D 8.5 B

二、填空题

- 8.6 【1】argc 【2】*argv[] 【3】argv[1] 【4】EOF 【5】fp

* 第 10 章 C++ 简介

10.1

```
#include <iostream.h>
void main()
{
    char c;
    char c1,c2;
    cout<<"Enter a char: ";
    cin>>c;
    c1=c-1;
    c2=c+1;
    cout <<c1<<" " <<(int)c1<<" " <<c<<" " <<(int)c
        <<" " <<c2<<" " <<(int)c2<<endl;
}
```

10.2

```
#include <iostream.h>
const float PI=3.14159f;
class CircleArea
{
    float radius;
public:
    void setRadius(float Radius);
    void getArea();
};
void CircleArea::setRadius(float Radius)
{
    radius=Radius;
}
void CircleArea::getArea()
{
```

```
    float area;  
    area=PI * radius * radius;  
    cout<<"Area of circle is "<<area<<endl;  
}  
void main()  
{  
    float r;  
    CircleArea circle;  
    cout<<"Enter radius of a circle: ";  
    cin>>r;  
    circle.setRadius(r);  
    circle.getArea();  
}
```

10.3

```
3  
3  
3
```

10.4

```
B::B(3)  
B::B()  
B::B(-3)  
B::B(3)
```

10.5

```
B::p  
B::q  
D::p  
D::q  
B::p  
B::q  
D::p  
B::q  
D::p  
D::q
```

10.6

```
5  
a
```

10.7

```
Class B  
Class C
```

10.8

```
#include <iostream. h>
#include <iomanip. h>
const int N=10;
class maxminswap
{
    public:
        maxminswap();
        void read();
        void result();
        void print();
    private:
        int i, min, max, maxd, mind, a[N];
};
maxminswap::maxminswap()
{
}
void maxminswap::read()
{
    for(i=0;i<N;i++)
        cin>>a[i];
    mind=maxd=0;
    min=a[0];
    max=a[0];
    return ;
}
void maxminswap::result()
{
    for(i=0;i<N;i++)
    {
        if(a[i]<min)
        {
            min=a[i];
            mind=i;
        }
        if(a[i]>max)
        {
            max=a[i];
            maxd=i;
        }
    }
}
```



```
    a[mind]=a[0];
    a[0]=min;
    a[maxd]=a[N-1];
    a[N-1]=max;
}
void maxminswap::print()
{
    for(i=0;i<N;i++)
        cout<<a[i]<<setw(4);
    cout<<endl;
}
void main()
{
    maxminswap a[N], *p;
    p=a;
    p->read();
    p->result();
    p->print();
}
```

10.9

```
#include <iostream.h>
#include <math.h>
class Polynomial
{
public:
    Polynomial();
    Polynomial(double x, double y, double z);
    void read();
    void roots()const;
private:
    double a,b,c;
};
Polynomial::Polynomial(double x, double y, double z)
{
    a=x;
    b=y;
    c=z;
}
void Polynomial::roots()const
{

```

```
double d, x1, x2, re, im;
d = b * b - 4 * a * c;
if (d >= 0)
{
    x1 = (-b + sqrt(d)) / (2 * a);
    x2 = (-b - sqrt(d)) / (2 * a);
    cout << "x1=" << x1 << " x2=" << x2 << endl;
}
else
{
    re = -b / (2 * a);
    im = sqrt(-d) / (2 * a);
    cout << "re=" << re << " im=" << im << endl;
}
}

void main()
{
    double x1, y1, z1;
    cout << "Please enter three numbers: ";
    cin >> x1 >> y1 >> z1;
    Polynomial value(x1, y1, z1);
    value.roots();
}
```